



ऑपरेटिंग सिस्टम

BTEUP CSE, तीसरे सेमेस्टर (विषय 3.1) के लिए एक संपूर्ण पाठ्यपुस्तक

KAMALDNP kamaldnp.com

इस किताब के बारे में

यह किताब प्राविधिक शिक्षा परिषद, उत्तर प्रदेश (BTEUP) के ऑपरेटिंग सिस्टम विषय — कंप्यूटर साइंस एवं इंजीनियरिंग, तीसरा सेमेस्टर, विषय 3.1 — को पहली परिभाषा से अंतिम फाइनल सिस्टम तक पढ़ाती है, आधिकारिक पाठ्यक्रम का इकाई दर इकाई और विषय दर विषय पालन करते हुए। हर अध्याय अपने विचारों को पहले सरल भाषा में समझाता है, आंकिक प्रश्नों को पूरी तरह हल करता है (CPU शेड्यूलिंग, बैंकर एल्गोरिदम, मेमोरी fit, पेजिंग, पेज प्रतिस्थापन, डिस्क शेड्यूलिंग), और अभ्यास के साथ समाप्त होता है। एक साथी प्रश्न बैंक — वस्तुनिष्ठ, लघु उत्तर और दीर्घ उत्तर प्रश्न उत्तरों सहित, हल किए गए आंकिक प्रश्न और बोर्ड पैटर्न पर दो मॉडल पेपर — इस किताब के साथ प्रकाशित है।

विषय-सूची

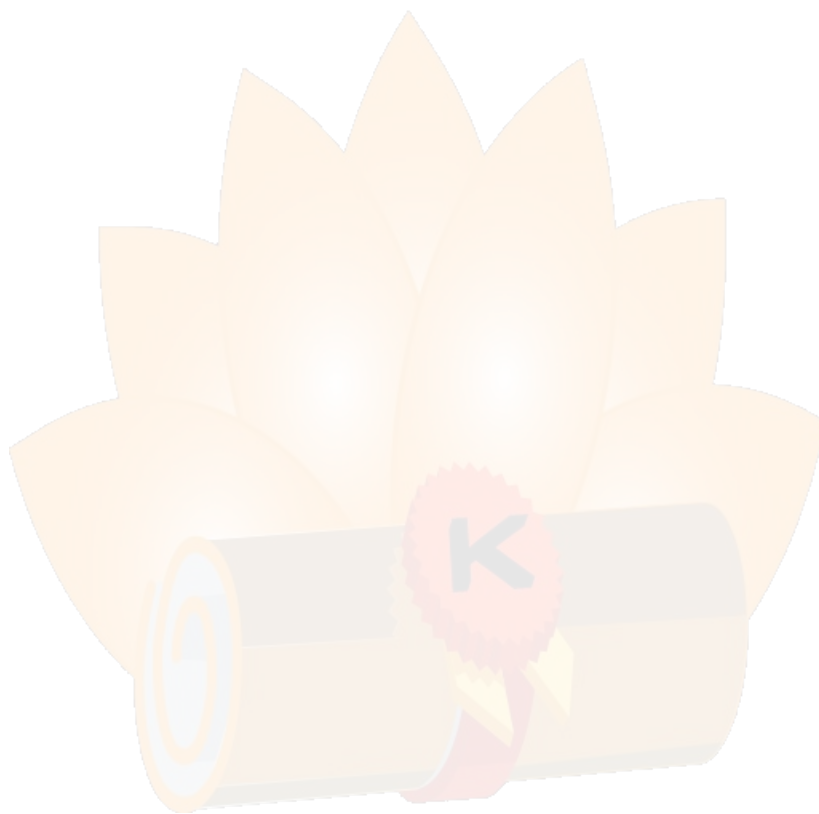
- **इकाई 1 — ऑपरेटिंग सिस्टम का परिचय:** अध्याय 1 परिचय और प्रकार · अध्याय 2 कार्य और सेवाएँ, कर्नेल, सिस्टम कॉल और सिस्टम प्रोग्राम
- **इकाई 2 — प्रक्रिया प्रबंधन:** अध्याय 3 प्रक्रिया अवधारणाएँ और कार्य · अध्याय 4 CPU शेड्यूलिंग · अध्याय 5 IPC, सिंक्रनाइज़ेशन और क्रिटिकल सेक्शन · अध्याय 6 डेडलॉक और बैंकर एल्गोरिदम
- **इकाई 3 — मेमोरी प्रबंधन:** अध्याय 7 मेमोरी आवंटन: First, Best और Worst Fit · अध्याय 8 स्वैपिंग, पेजिंग और सेगमेंटेशन · अध्याय 9 पेज फ़ॉल्ट, पेज प्रतिस्थापन, विखंडन और वर्चुअल मेमोरी
- **इकाई 4 — I/O सिस्टम:** अध्याय 10 मास स्टोरेज और डिस्क संरचना · अध्याय 11 डिस्क शेड्यूलिंग, स्वैप स्थान और RAID

- **इकाई 5 — फाइल प्रबंधन:** अध्याय 12 फाइल अवधारणा, एक्सेस विधियाँ, डायरेक्टरी संरचना और माउंटिंग · अध्याय 13 फाइल साझा करना और सुरक्षा, कार्यान्वयन और प्रकार
- **परिशिष्ट A** — ऑपरेटिंग सिस्टम लैब: 12 प्रयोग

इस किताब का उपयोग कैसे करें

अध्यायों को क्रम में पढ़िए; हर इकाई पिछली पर बनती है। हर खंड के अंदर का छोटा अभ्यास सेट आगे पढ़ने से पहले कीजिए, अध्याय अभ्यास सेट को कागज़ पर हल कीजिए, और फिर प्रश्न बैंक से खुद को परखिए। हल किए गए आंकिक प्रश्न विधि के रूप में नकल करने के लिए लिखे गए हैं: पहले Gantt चार्ट, फिर सूत्र, अंत में उत्तर।

KAMALDNP द्वारा प्रकाशित · kamaldnp.com · support@kamaldnp.com





अध्याय 1: परिचय और ऑपरेटिंग सिस्टम के प्रकार

(BTEUP CSE तीसरा सेमेस्टर · विषय 3.1 · इकाई 1 — ऑपरेटिंग सिस्टम का परिचय)

आप क्या सीखेंगे

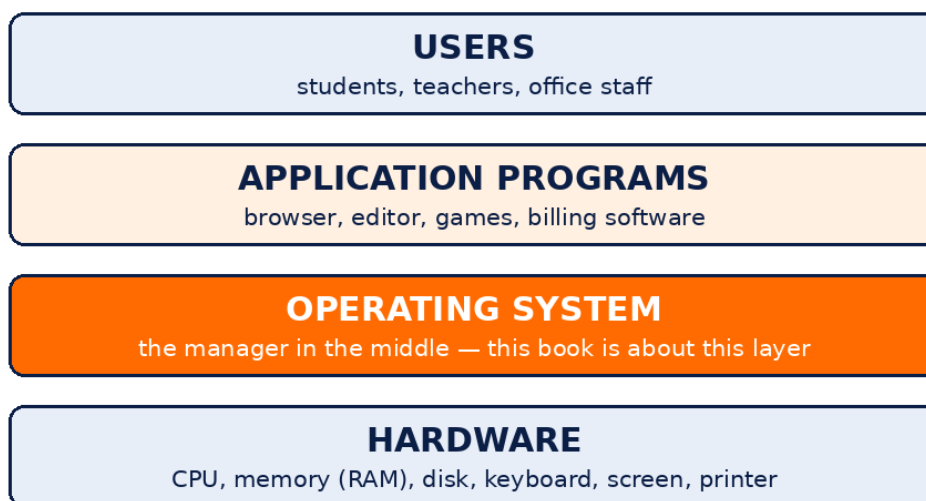
- ऑपरेटिंग सिस्टम क्या है — एक पूरी परिभाषा जो एक बार पढ़ने में समझ आ जाए
- कंप्यूटर इसके बिना उपयोगी ढंग से क्यों नहीं चल सकता
- ऑपरेटिंग सिस्टम के दो मुख्य लक्ष्य
- ऑपरेटिंग सिस्टम कैसे विकसित हुए, संक्षेप में
- मुख्य प्रकार: बैच, मल्टीप्रोग्राम्ड, मल्टीटास्किंग और टाइम-शेयरिंग, मल्टीप्रोसेसिंग, रियल-टाइम, डिस्ट्रिब्यूटेड और एम्बेडेड — और उनमें अंतर

1.1 ऑपरेटिंग सिस्टम क्या है?

ऑपरेटिंग सिस्टम (OS) वह सिस्टम सॉफ्टवेयर है जो उपयोगकर्ता (और एप्लिकेशन प्रोग्रामों) तथा कंप्यूटर हार्डवेयर के बीच की परत का काम करता है: यह पूरी मशीन को नियंत्रित और संचालित करता है, उसके सारे संसाधनों — प्रोसेसर (CPU), मेमोरी, स्टोरेज डिस्क और इनपुट/आउटपुट उपकरणों — का प्रबंधन करता है, और वे सेवाएँ और वातावरण प्रदान करता है जिनकी एप्लिकेशन प्रोग्रामों को चलने के लिए ज़रूरत होती है। हर प्रोग्राम, ब्राउज़र से लेकर खेल तक, ऑपरेटिंग सिस्टम के अधीन चलता है; उपयोगकर्ता कभी हार्डवेयर से सीधे नहीं निपटता।

एक बड़े रेस्तरां की कल्पना कीजिए। ग्राहक कभी रसोई में नहीं जाते, और रसोइये मेज़ों से ऑर्डर नहीं लेते। बीच में मैनेजर और वेटर खड़े होते हैं: ऑर्डर लेते हैं, तय करते हैं कि कौन सा रसोइया क्या बनाएगा, देखते हैं कि दो ऑर्डर आपस में न टकराएँ, और सामग्री का हिसाब रखते हैं। ऑपरेटिंग सिस्टम कंप्यूटर के लिए वही मैनेजर है — हार्डवेयर रसोई है, आपके प्रोग्राम ग्राहकों के ऑर्डर हैं, और OS सब कुछ क्रमबद्ध ढंग से चलाता रहता है।

Where the Operating System Sits



Requests flow down; results flow up. Users never touch hardware directly.

चित्र 1.1 - ऑपरेटिंग सिस्टम उपयोगकर्ताओं, प्रोग्रामों और हार्डवेयर के बीच की परत है।

उदाहरण की समझ: बीच की परत न हो, तो हर एप्लिकेशन को खुद हर डिस्क, प्रिंटर और मेमोरी चिप चलानी आनी चाहिए — और दो प्रोग्राम एक ही समय पर एक ही संसाधन के लिए लड़ सकते हैं। OS इसलिए है ताकि हार्डवेयर का इस्तेमाल **सुविधाजनक** (उपयोगकर्ता के लिए आसान) और **कुशल** (कुछ बर्बाद न हो) ढंग से हो। यही इसके दो लक्ष्य हैं, और इस किताब का लगभग हर विषय इन्हीं दो लक्ष्यों में से एक का काम करता हुआ रूप है:

- **सुविधा** — उपयोगकर्ता मशीन के अंदर की बातें जाने बिना प्रोग्राम चला सके, फाइल सहेज सके और प्रिंट कर सके।
- **कुशलता** — महँगा हार्डवेयर (CPU, RAM) शायद ही कभी खाली बैठे; एक प्रोग्राम कीबोर्ड का इंतज़ार कर रहा हो, तो CPU को किसी और का काम करना चाहिए।

अभ्यास सेट 1.1

- एप्लिकेशन प्रोग्रामों और हार्डवेयर के बीच कौन सी परत होती है?
- एक कंप्यूटर में हार्डवेयर और एप्लिकेशन प्रोग्राम हैं लेकिन OS नहीं है। तुरंत आने वाली दो समस्याएँ बताइए।
- "OS एक संसाधन प्रबंधक (resource manager) है।" रेस्तरां वाले उदाहरण से इस लाइन को समझाइए।
- जब पहला प्रोग्राम प्रिंटआउट का इंतज़ार करता है, तो CPU दूसरा प्रोग्राम शुरू कर देता है — इससे OS का कौन सा लक्ष्य पूरा होता है, सुविधा या कुशलता? क्यों?

1.2 ऑपरेटिंग सिस्टम कैसे विकसित हुए — एक छोटा इतिहास

ऑपरेटिंग सिस्टम एक दिन में नहीं बने; हर पीढ़ी ने पिछली पीढ़ी की एक तकलीफ़ दूर की।

- **कोई OS नहीं (1945-1950 के मध्य तक)**। प्रोग्राम स्विच और पंच कार्ड से हाथ से डाले जाते थे; प्रोग्रामर ही ऑपरेटर था। अगला प्रोग्राम तैयार होते समय मशीन खाली खड़ी रहती थी।
- **बैच सिस्टम (1950 के मध्य-1960 के दशक)**। एक जैसे कामों को एक बैच (समूह) में इकट्ठा करके एक के बाद एक अपने आप चलाया जाता था। कामों के बीच का खाली समय घटा, लेकिन प्रोग्रामर को आउटपुट के लिए घंटों इंतज़ार करना पड़ता था और चलते काम से बातचीत नहीं हो सकती थी।
- **मल्टीप्रोग्रामिंग (1960 का दशक)**। कई काम एक साथ मेमोरी में रखे जाते थे; जब एक काम इनपुट/आउटपुट का इंतज़ार करता, तो CPU दूसरे पर चला जाता था। CPU का खाली समय तेज़ी से घटा — यह कुशलता वाले लक्ष्य की जीत है।
- **टाइम-शेयरिंग / मल्टीटास्किंग (1960-70 के दशक से आगे)**। CPU कामों के बीच इतनी तेज़ी से बदलता है (समय के छोटे टुकड़ों में) कि कई उपयोगकर्ताओं को लगता है कि मशीन सिर्फ़ उनकी है। बातचीत संभव हो गई — यह सुविधा वाले लक्ष्य की जीत है।
- **पर्सनल कंप्यूटर OS (1980 का दशक)**। एक उपयोगकर्ता, एक मशीन: पहले कमांड-लाइन सिस्टम (MS-DOS), फिर ग्राफ़िकल वाले (Windows, macOS) — इस्तेमाल की आसानी मुख्य उद्देश्य बन गई।
- **आधुनिक सिस्टम (1990 के दशक-आज तक)**। नेटवर्किंग, सुरक्षा, कई कोर, मोबाइल और क्लाउड सिस्टम — आज का OS (Linux, Windows, Android, iOS) सारे पुराने विचारों का मेल है, कहीं ज़्यादा सुरक्षा और परिष्कार के साथ।

अभ्यास सेट 1.2

- इन्हें विकास के क्रम में लगाइए: टाइम-शेयरिंग, बैच, कोई OS नहीं, मल्टीप्रोग्रामिंग।
- किस पीढ़ी ने पहली बार बातचीत वाली (interactive) कंप्यूटिंग संभव की?
- मल्टीप्रोग्रामिंग ने मुख्य रूप से दो लक्ष्यों में से किसे सुधारा? एक लाइन में कारण दीजिए।

1.3 ऑपरेटिंग सिस्टम के प्रकार

पाठ्यक्रम ऑपरेटिंग सिस्टम के विभिन्न प्रकार पूछता है। नीचे का हर प्रकार ऊपर के इतिहास की एक पीढ़ी से निकला है, और हर एक आज भी कहीं न कहीं इस्तेमाल में है।

1.3.1 बैच ऑपरेटिंग सिस्टम

एक प्रकार के कामों को एक बैच में इकट्ठा करके, चलने के दौरान उपयोगकर्ता से किसी बातचीत के बिना, एक के बाद एक चलाया जाता है; उपयोगकर्ता काम जमा करता है, इंतज़ार करता है, और बाद में आउटपुट ले लेता है। वहाँ उपयुक्त है जहाँ बड़े डेटा पर एक ही प्रक्रिया बार-बार दोहराई जाती है — पुरानी वेतन (payroll) और बिलिंग प्रणालियाँ इसका क्लासिक उदाहरण हैं। कमज़ोरी: कोई बातचीत नहीं, और एक काम की गलती या देरी पूरी कतार को प्रभावित करती है।

1.3.2 मल्टीप्रोग्राम्ड ऑपरेटिंग सिस्टम

कई प्रोग्राम एक ही समय मुख्य मेमोरी में रखे जाते हैं, और जब भी चलता हुआ प्रोग्राम इनपुट/आउटपुट का इंतज़ार करता है, CPU को दूसरे तैयार प्रोग्राम पर ले जाया जाता है, ताकि CPU लगभग हमेशा व्यस्त रहे। यह कुशलता वाले लक्ष्य की बुनियाद है; ज़्यादातर आधुनिक सिस्टम इसी विचार के वंशज हैं।

1.3.3 मल्टीटास्किंग और टाइम-शेयरिंग ऑपरेटिंग सिस्टम

मल्टीटास्किंग OS कई प्रोग्रामों को CPU उनके बीच बहुत तेज़ी से बदलकर उपयोगकर्ता की दृष्टि से "एक ही समय पर" चलाता है; टाइम-शेयरिंग सिस्टम इसका बहु-उपयोगकर्ता रूप है, जहाँ अलग-अलग टर्मिनलों पर कई उपयोगकर्ता एक मशीन साझा करते हैं, हर एक को बारी-बारी से समय के छोटे टुकड़े मिलते हैं, ताकि सबको त्वरित जवाब मिले। आपका Windows/Linux डेस्कटॉप मल्टीटास्किंग है; किसी कॉलेज का वह सर्वर जिसे पूरी कक्षा टर्मिनलों से इस्तेमाल करती है, वह टाइम-शेयरिंग है।

1.3.4 मल्टीप्रोसेसिंग ऑपरेटिंग सिस्टम

दो या अधिक CPU (या कोर) वाला सिस्टम जो मेमोरी साझा करते हैं और प्रोग्रामों पर सच में समानांतर काम करते हैं — बदल-बदल कर नहीं, बल्कि सचमुच एक साथ। वहाँ इस्तेमाल होता है जहाँ गति और विश्वसनीयता मायने रखती है: सर्वर, वर्कस्टेशन, और हर आधुनिक मल्टी-कोर PC और फ़ोन।

1.3.5 रियल-टाइम ऑपरेटिंग सिस्टम (RTOS)

वह OS जो कड़ाई से तय, गारंटीशुदा समय सीमा के भीतर जवाब देने के लिए बना हो; देर होना असफलता माना जाता है। **हार्ड रियल-टाइम** सिस्टम कभी कोई समय-सीमा नहीं चूकते (उड़ान नियंत्रण, पेसमेकर, औद्योगिक सुरक्षा)। **सॉफ़्ट रियल-टाइम** सिस्टम पूरी कोशिश करते हैं लेकिन कभी-कभार की देरी चल जाती है (वीडियो स्ट्रीमिंग, ऑनलाइन खेल)। Windows जैसे साधारण OS रियल-टाइम सिस्टम नहीं हैं — वे कभी ठीक जवाब के क्षण का वादा नहीं करते।

1.3.6 डिस्ट्रिब्यूटेड ऑपरेटिंग सिस्टम

वह OS जो नेटवर्क से जुड़े कई स्वतंत्र कंप्यूटरों पर चलता है और उन्हें उपयोगकर्ता के लिए एक ही सिस्टम जैसा बनाता है — फाइलें, प्रोसेसिंग और उपकरण मशीनों के बीच साझा होते हैं। बड़ी वेब सेवाएँ और क्लस्टर इसी तरह काम करते हैं।

1.3.7 एम्बेडेड ऑपरेटिंग सिस्टम

किसी उपकरण के अंदर बना एक छोटा, विशेष OS जो एक ही निश्चित काम करता है — वॉशिंग मशीन, माइक्रोवेव ओवन, ATM मशीन, स्मार्ट TV, कार के नियंत्रक। यह छोट्टा हुआ होता है: तेज़ी से शुरू होने वाला, कम मेमोरी वाला, अक्सर रियल-टाइम प्रकृति का।

| प्रकार | एक लाइन की पहचान | आम उदाहरण | |---|---|---| | बैच | काम समूहों में, कोई बातचीत नहीं | पुरानी वेतन/बिलिंग प्रोसेसिंग | | मल्टीप्रोग्राम्ड | कई काम मेमोरी में, CPU कभी खाली नहीं | शुरुआती मेनफ्रेम सिस्टम | | मल्टीटास्किंग / टाइम-शेयरिंग | तेज़ बदलाव; कई उपयोगकर्ताओं को अपना होने का एहसास | Windows डेस्कटॉप; कॉलेज सर्वर | | मल्टीप्रोसेसिंग | कई CPU, सच में समानांतर | सर्वर, मल्टी-कोर PC | | रियल-टाइम | गारंटीशुदा जवाब का समय | उड़ान नियंत्रण, पेसमेकर | | डिस्ट्रिब्यूटेड | कई कंप्यूटर एक जैसे काम करते हैं | क्लस्टर, बड़ी वेब सेवाएँ | | एम्बेडेड | एक काम के लिए उपकरण के अंदर बना | वॉशिंग मशीन, ATM |

अभ्यास सेट 1.3

- ATM को अनुमानित जवाब देना होता है और एक निश्चित काम करना होता है। इसके OS को कौन से दो प्रकार बताते हैं, और क्यों?
- "टाइम-शेयरिंग सिस्टम मल्टीप्रोग्रामिंग का तार्किक विस्तार है।" दो लाइनों में समझाइए।
- अस्पताल का मॉनिटर कभी देर से जवाब नहीं दे सकता। बैच या रियल-टाइम — कौन सा, और किस तरह का (हार्ड/सॉफ़्ट)?

- मिलाइए: (a) 1970 की वेतन सूची
(b) कॉलेज लैब सर्वर
(c) स्मार्ट TV — बैच, टाइम-शेयरिंग, एम्बेडेड से।

आम गलतियाँ

- OS को एप्लिकेशन प्रोग्राम कहना — यह सिस्टम सॉफ्टवेयर है, वह परत जिस पर एप्लिकेशन चलते हैं।
- मल्टीटास्किंग और मल्टीप्रोसेसिंग में भ्रम: एक CPU पर तेज़ी से बदलना, कई CPU के सच में समानांतर काम करने जैसा नहीं है।
- रियल-टाइम का मतलब "बहुत तेज़" कहना। इसका मतलब है समय पर होने की गारंटी — समय-सीमा का वादा, कच्ची गति नहीं।
- OS के कार्य पूछे जाने पर Windows की विशेषताएँ गिनाना — कार्य प्रक्रिया और मेमोरी प्रबंधन जैसे काम हैं (अध्याय 2), उत्पाद की विशेषताएँ नहीं।

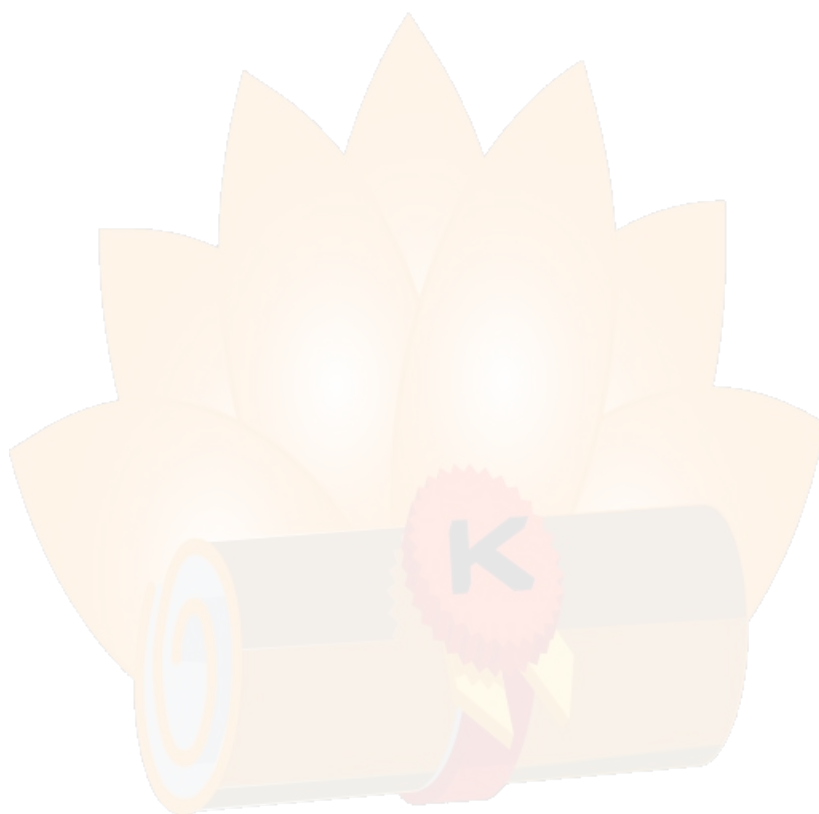
सारांश

- OS उपयोगकर्ताओं/प्रोग्रामों और हार्डवेयर के बीच की प्रबंधक परत है; इसके लक्ष्य सुविधा और कुशलता हैं।
- इतिहास इस क्रम में बढ़ा: कोई OS नहीं → बैच → मल्टीप्रोग्रामिंग → टाइम-शेयरिंग → पर्सनल और आधुनिक सिस्टम, हर चरण खाली समय या बातचीत की एक तकलीफ़ दूर करता हुआ।
- प्रकार: बैच, मल्टीप्रोग्राम्ड, मल्टीटास्किंग/टाइम-शेयरिंग, मल्टीप्रोसेसिंग, रियल-टाइम (हार्ड/सॉफ्ट), डिस्ट्रिब्यूटेड, एम्बेडेड — इस आधार पर वर्गीकृत कि काम कैसे संचालित होता है और किस बात की गारंटी दी जाती है।

अध्याय अभ्यास सेट

- ऑपरेटिंग सिस्टम को एक पूरे वाक्य में परिभाषित कीजिए, यह बताते हुए कि यह किसका प्रबंधन करता है और क्या प्रदान करता है।
- OS के दो लक्ष्य बताइए और हर एक के लिए एक उदाहरण स्थिति दीजिए।
- मल्टीटास्किंग और मल्टीप्रोसेसिंग में अंतर बताइए (दो बिंदु)।
- बैच और टाइम-शेयरिंग सिस्टम में अंतर बताइए (दो बिंदु)।

- हार्ड रियल-टाइम सिस्टम क्या है? दो उदाहरण दीजिए।
- मल्टीप्रोग्रामिंग ने CPU का खाली समय क्यों घटाया? दो कामों के छोटे उदाहरण से समझाइए।
- आप किस प्रकार का OS चुनेंगे: (a) एक साथ हजारों लोग इस्तेमाल करें ऐसा रेलवे आरक्षण सर्वर, (b) वॉशिंग मशीन, (c) मिसाइल मार्गदर्शन प्रणाली? हर एक का एक लाइन में औचित्य दीजिए।
- OS के विकास को चार चरणों में लिखिए, हर चरण का मुख्य सुधार बताते हुए।





अध्याय 2: कार्य और सेवाएँ, कर्नेल, सिस्टम कॉल और सिस्टम प्रोग्राम

(BTEUP CSE तीसरा सेमेस्टर · विषय 3.1 · इकाई 1 — ऑपरेटिंग सिस्टम का परिचय)

आप क्या सीखेंगे

- ऑपरेटिंग सिस्टम के मुख्य कार्य (काम) — वह सूची जिसे यह पूरी किताब विस्तार देती है
- OS द्वारा प्रोग्रामों और उपयोगकर्ताओं को दी जाने वाली सेवाएँ
- कर्नेल क्या है, और कर्नेल को संगठित करने के दो तरीके (मोनोलिथिक और माइक्रोकर्नेल)
- सिस्टम कॉल क्या है, वह कैसे यात्रा करती है, और उसके मुख्य प्रकार
- UNIX/Linux संरचना, परत दर परत
- सिस्टम प्रोग्राम क्या हैं, रोज़मर्रा के उदाहरणों के साथ

2.1 ऑपरेटिंग सिस्टम के कार्य

अध्याय 1 ने OS को प्रबंधक कहा था। प्रबंधन पाँच मुख्य कार्यों में बँटता है — इन्हें अच्छी तरह सीखिए, क्योंकि इस पाठ्यक्रम की इकाई 2 से 5 इन्हीं पाँच कार्यों का एक-एक करके अध्ययन हैं:

- **प्रक्रिया प्रबंधन (Process management)** — चल रहे प्रोग्रामों (प्रक्रियाओं) को बनाना, क्रम तय करना और समाप्त करना, और उन्हें CPU का निष्पक्ष साझा करने देना। (इकाई 2)
- **मेमोरी प्रबंधन (Memory management)** — तय करना कि किस प्रक्रिया को RAM का कौन सा हिस्सा मिलेगा, और हर प्रक्रिया की मेमोरी को अलग और सुरक्षित रखना। (इकाई 3)
- **फाइल प्रबंधन (File management)** — डिस्क पर डेटा को फाइलों और डायरेक्ट्रियों में संगठित करना, और नियंत्रित करना कि कौन उन्हें पढ़ या बदल सकता है। (इकाई 5)
- **उपकरण (I/O) प्रबंधन (Device management)** — डिवाइस ड्राइवों के द्वारा प्रिंटर, डिस्क, कीबोर्ड और स्क्रीन को नियंत्रित करना, ताकि प्रोग्रामों को हार्डवेयर का ब्योरा न जानना पड़े। (इकाई 4)

- **सुरक्षा और संरक्षण (Protection and security)** — यह सुनिश्चित करना कि उपयोगकर्ता और प्रोग्राम केवल वही कर सकें जिसकी उन्हें अनुमति है, और सिस्टम या एक-दूसरे के काम को नुकसान न पहुँचा सकें। (सभी इकाइयों में चलता है)

इनके आसपास सहायक काम हैं: त्रुटि पहचान (हार्डवेयर या प्रोग्राम की खराबियों पर नज़र रखना), लेखा-जोखा (किसने कितना CPU और मेमोरी इस्तेमाल किया, इसका रिकॉर्ड), और उपयोगकर्ता इंटरफ़ेस (कमांड लाइन या GUI) जिसके द्वारा लोग निर्देश देते हैं।

अभ्यास सेट 2.1

- एक छात्र दो प्रोग्राम खोलता है और दोनों को एक-दूसरे को बिगाड़े बिना RAM मिलती है। कौन सा कार्य काम कर रहा है?
- मिलाइए: (a) तय करना कि अगला प्रोग्राम कौन सा चलेगा
(b) फाइल को दूसरे उपयोगकर्ताओं से सुरक्षित रखना
(c) प्रिंटर से बात करना — प्रक्रिया, सुरक्षा, उपकरण प्रबंधन से।
- उपयोगकर्ता इंटरफ़ेस को पाँच मुख्य संसाधन कार्यों में क्यों नहीं गिना जाता?

2.2 ऑपरेटिंग सिस्टम की सेवाएँ

OS दो दिशाओं में सेवाएँ देता है। **प्रोग्रामों के लिए:** प्रोग्राम निष्पादन (प्रोग्राम लोड करके चलाना), इनपुट/आउटपुट कार्य (उनकी ओर से पढ़ना/लिखना, क्योंकि प्रोग्राम उपकरणों को सीधे नहीं छू सकते), फाइल-सिस्टम कार्य (फाइल बनाना, मिटाना, पढ़ना, लिखना), प्रक्रियाओं के बीच संचार, और त्रुटि सूचना।

उपयोगकर्ताओं के लिए: एक उपयोगकर्ता इंटरफ़ेस (CLI या GUI), और रोज़मर्रा के कामों के लिए सिस्टम प्रोग्राम (अगला खंड) जैसे फाइल कॉपी करना और डिस्क की सामग्री देखना। याद रखने का उपयोगी तरीका: कार्य OS की ज़िम्मेदारी है; सेवा वह ज़िम्मेदारी है जैसा उसे कोई प्रोग्राम या उपयोगकर्ता अनुभव करता है।

अभ्यास सेट 2.2

- जब आपका प्रोग्राम `print()` बुलाता है, तो वह कौन सी दो सेवाएँ इस्तेमाल करता है?
- "त्रुटि पहचान" — यह कार्य है, सेवा है, या दोनों? एक लाइन में अपने उत्तर का बचाव कीजिए।

2.3 कर्नेल — OS का हृदय

कर्नेल ऑपरेटिंग सिस्टम का वह केंद्रीय भाग है जो कंप्यूटर चालू रहने तक हर समय मेमोरी में लोड रहता है और एकमात्र ऐसा भाग है जिसे हार्डवेयर से सीधे काम करने की अनुमति है: यह केंद्रीय दायित्व निभाता है — प्रक्रियाओं का क्रम तय करना, मेमोरी का प्रबंधन, इंटरप्ट सँभालना और सिस्टम कॉल को पूरा करना। OS का बाकी सब कुछ इसके इर्द-गिर्द बना होता है।

कर्नेल दो क्लासिक तरीकों से संगठित होते हैं:

- **मोनोलिथिक कर्नेल** — सारी OS सेवाएँ (प्रक्रिया, मेमोरी, फाइल, उपकरण प्रबंधन) कर्नेल के अंदर एक बड़े प्रोग्राम के रूप में साथ चलती हैं। तेज़, क्योंकि भाग सीधे बात करते हैं; लेकिन किसी भी भाग की खराबी पूरे सिस्टम को गिरा सकती है, और कोड का रखरखाव बड़ा होता है। Linux इसी डिज़ाइन का पालन करता है।
- **माइक्रोकर्नेल** — केवल न्यूनतम आवश्यक भाग (बुनियादी क्रम-निर्धारण, मेमोरी की बुनियाद, संदेश पहुँचाना) कर्नेल में रहता है; बाकी सेवाएँ बाहर साधारण प्रोग्रामों के रूप में चलती हैं और संदेशों से बात करती हैं। अधिक विश्वसनीय (कोई सेवा कर्नेल को मारे बिना क्रैश हो सकती है) और विस्तार में आसान, लेकिन संदेश पहुँचाने की वजह से धीमा। MINIX इसका मानक पाठ्यपुस्तक उदाहरण है।

आधार	मोनोलिथिक कर्नेल	माइक्रोकर्नेल	---	---	---	अंदर क्या चलता है	सारी OS सेवाएँ
केवल न्यूनतम केंद्रीय सेवाएँ	गति	तेज़ (सीधी कॉल)	धीमा (संदेश पहुँचाना)	विश्वसनीयता		एक खराबी सब कुछ गिरा सकती है	सेवा का क्रैश कर्नेल को गिराए बिना हो सकता है
आकार /	रखरखाव	बड़ा, कठिन	छोटा केंद्र, विस्तार आसान	उदाहरण	Linux	MINIX	

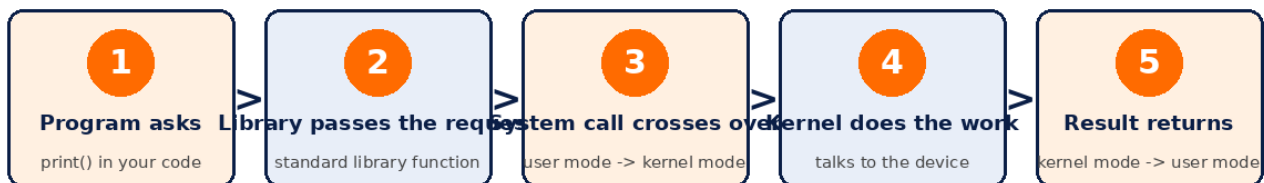
अभ्यास सेट 2.3

- कर्नेल को हर समय मेमोरी में क्यों रहना चाहिए?
- एक ड्राइवर क्रैश होता है और पूरा सिस्टम उसी के साथ अटक जाता है। यह किस कर्नेल डिज़ाइन की ओर इशारा करता है, और क्यों?
- माइक्रोकर्नेल अधिक विश्वसनीय हैं फिर भी Linux मोनोलिथिक है — बदले में Linux कौन सी विशेषता पाता है?

2.4 सिस्टम कॉल — प्रोग्राम कर्नेल से कैसे माँगते हैं

सिस्टम कॉल वह नियंत्रित द्वार है जिसके द्वारा कोई चलता हुआ प्रोग्राम कर्नेल से सेवा का अनुरोध करता है — जैसे फाइल पढ़ना, स्क्रीन पर लिखना, प्रक्रिया बनाना या मेमोरी आवंटित करना — क्योंकि प्रोग्राम प्रतिबंधित यूज़र मोड में चलते हैं और उन्हें हार्डवेयर या सुरक्षित मेमोरी को सीधे छूने की अनुमति नहीं होती; अनुरोध CPU को विशेषाधिकार प्राप्त कर्नेल मोड में ले जाता है, कर्नेल जाँच करके काम करता है, और परिणाम लौटाता है। यही पैटर्न आप Python किताब में हर बार मिले थे जब कोई प्रोग्राम प्रिंट करता था या फाइल खोलता था — वही द्वार हर भाषा की सेवा करता है।

How a System Call Works



The kernel is the only part allowed to touch hardware directly.
That is why every request must pass through a system call.

चित्र 2.1 - सिस्टम कॉल प्रोग्राम से लाइब्रेरी होते हुए कर्नेल तक जाती है और वापस आती है।

सिस्टम कॉल के मुख्य प्रकार (मानक पाँच समूह):

- **प्रक्रिया नियंत्रण** — प्रक्रिया बनाना/समाप्त करना, प्रोग्राम लोड और चलाना, प्रक्रिया की प्रतीक्षा करना (UNIX में उदाहरण: `fork`, `exit`, `wait`)।
- **फाइल प्रबंधन** — फाइल बनाना, मिटाना, खोलना, बंद करना, पढ़ना, लिखना (`open`, `read`, `write`, `close`)।
- **उपकरण प्रबंधन** — उपकरण माँगना/छोड़ना, उपकरण पढ़ना/लिखना (`ioctl`, उपकरणों पर `read/write`)।
- **सूचना रखरखाव** — समय, तारीख, प्रक्रिया और सिस्टम की जानकारी पाना/रखना (`getpid`, `time`)।
- **संचार** — कनेक्शन बनाना/मिटाना, संदेश भेजना/पाना, साझा मेमोरी (`pipe`, `shmget`)।

प्रोग्राम शायद ही कभी इन्हें सीधे बुलाता है; वह लाइब्रेरी फंक्शन (जैसे printf या print) बुलाता है, और लाइब्रेरी असली सिस्टम कॉल करती है। एक लाइब्रेरी कॉल का मतलब ठीक एक सिस्टम कॉल हो सकता है — या कई, या कोई भी नहीं अगर जवाब लाइब्रेरी के बफ़र में पहले से हो।

अभ्यास सेट 2.4

- यूज़र मोड में कोई प्रोग्राम खुद डिस्क क्यों नहीं पढ़ सकता?
- पाँच प्रकारों में वर्गीकृत कीजिए: (a) नई प्रक्रिया बनाना
(b) फाइल पढ़ना
(c) वर्तमान समय पूछना
(d) दो प्रोग्रामों के बीच pipe खोलना।
- एक प्रोग्राम 100 लाइनों प्रिंट करता है लेकिन डिस्क/स्क्रीन का काम केवल कुछ ही बार होता है। इस खंड का कौन सा विचार इसे समझाता है?

2.5 UNIX/Linux संरचना

UNIX और उसका वंशज Linux OS को साफ़ परतों में संगठित करते हैं — पाठ्यक्रम इस संरचना को खास तौर पर नाम देता है, और अध्याय 1 की परतदार तस्वीर यहाँ ठोस बन जाती है:

- **हार्डवेयर** (सबसे नीचे) — CPU, मेमोरी, डिस्क, टर्मिनल।
- **कर्नेल** — केंद्र (खंड 2.3): प्रक्रिया, मेमोरी, फाइल और उपकरण प्रबंधन, जहाँ केवल सिस्टम कॉल के द्वारा पहुँचा जाता है।
- **सिस्टम कॉल इंटरफ़ेस** — खंड 2.4 का नियंत्रित द्वार।
- **लाइब्रेरी और उपयोगिताएँ** — मानक लाइब्रेरी फंक्शन, शेल और कमांड, कंपाइलर: वह परत जिसे प्रोग्राम और उपयोगकर्ता सच में छूते हैं।
- **उपयोगकर्ता और एप्लिकेशन प्रोग्राम** (सबसे ऊपर) — लोग और उनके द्वारा चलाया जाने वाला सॉफ़्टवेयर।

Linux मोनोलिथिक कर्नेल के साथ इस संरचना का पालन करता है (खंड 2.3), और **शेल** जोड़ता है — वह कमांड इंटरप्रेटर जो आपकी टाइप की गई कमांड (ls, cp, mkdir — OS Lab के पहले प्रयोग की वही कमांडें) लेता है और सिस्टम प्रोग्रामों के द्वारा कर्नेल से उन्हें पूरा करने को कहता है। Lab की Linux कमांडें और यह संरचना एक ही कहानी के दो पक्ष हैं।

अभ्यास सेट 2.5

- UNIX परत तस्वीर में, उपयोगकर्ता प्रोग्राम किस परत को कभी नहीं छोड़ सकता, और क्यों?
- शेल कहाँ बैठता है — कर्नेल के अंदर या बाहर? यह क्या करता है?

2.6 सिस्टम प्रोग्राम

सिस्टम प्रोग्राम ऑपरेटिंग सिस्टम के साथ दिए गए तैयार उपयोगिता प्रोग्राम हैं जो उसकी सेवाओं का इस्तेमाल करके उपयोगकर्ता के रोज़मर्रा के काम करते हैं — फाइल कार्य (कॉपी, स्थानांतरण, मिटाना, खोजना), स्थिति जानकारी (डिस्क स्थान, चलती प्रक्रियाएँ), प्रोग्राम विकास सहायता (कंपाइलर, लोडर, डिबगर), पाठ संपादन, और संचार (मेल, रिमोट लॉगिन)। ये न कर्नेल हैं न उपयोगकर्ता एप्लिकेशन: ये OS का अपना टूलबॉक्स हैं, दोनों के बीच बैठे हुए। Linux पर cp, mv, ls, grep जैसी कमांडें और शेल खुद सिस्टम प्रोग्राम हैं; Windows पर File Explorer, Task Manager और Command Prompt वही भूमिका निभाते हैं।

परीक्षा के लिए मायने रखने वाला अंतर: **कर्नेल** एक हमेशा चलने वाला केंद्र है; **सिस्टम कॉल** उसमें जाने का द्वार हैं; **सिस्टम प्रोग्राम** बाहर खड़े साधारण (बदले जा सकने वाले) प्रोग्राम हैं, जो उस द्वार का इस्तेमाल किसी भी अन्य प्रोग्राम की तरह करते हैं।

अभ्यास सेट 2.6

- वर्गीकृत कीजिए: कर्नेल, सिस्टम कॉल, सिस्टम प्रोग्राम — (a) cp कमांड
(b) read()
(c) शेड्यूलर
(d) OS के साथ दिया गया पाठ संपादक।
- यदि कोई सिस्टम प्रोग्राम मिटा दिया जाए, तो क्या OS काम करना बंद कर देता है? ठीक-ठीक क्या खोता है?

आम गलतियाँ

- इस अध्याय के तीन K-शब्दों को मिलाना: **कर्नेल** (केंद्र), **सिस्टम कॉल** (द्वार), **सिस्टम प्रोग्राम** (बाहर की उपयोगिता)। इन्हें अलग रखिए और इस अध्याय के आधे अंक सुरक्षित हैं।
- यह कहना कि उपयोगकर्ता इंटरफ़ेस ही OS है। GUI/शेल बदली जा सकने वाली बाहरी परत है; OS उसके बिना भी चल सकता है (सर्वर चलते हैं)।
- माइक्रोकर्नेल को हर तरह से "बेहतर" कहना — वे विश्वसनीयता के बदले गति देते हैं; परीक्षा के उत्तरों में यह समझौता बताना चाहिए।
- यह लिखना कि प्रोग्राम "सिस्टम कॉल के द्वारा हार्डवेयर को बुलाते हैं"। सिस्टम कॉल कर्नेल तक पहुँचती हैं; कर्नेल हार्डवेयर तक पहुँचता है।

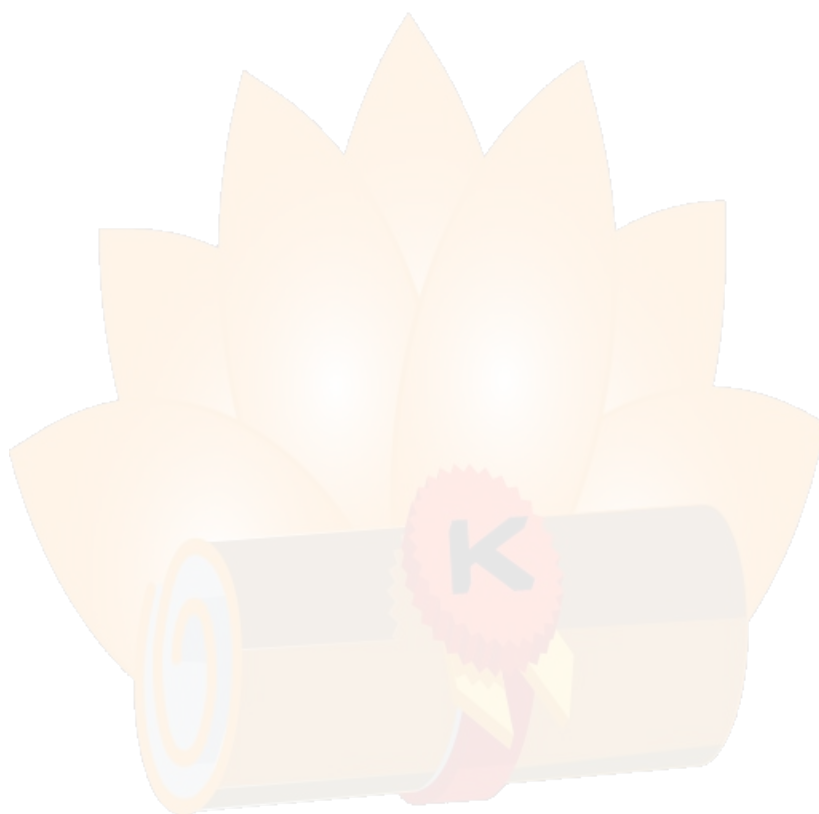
सारांश

- पाँच कार्य: प्रक्रिया, मेमोरी, फाइल, उपकरण प्रबंधन, और सुरक्षा/संरक्षण — इकाई 2-5 का नक्शा।
- सेवाएँ वही कार्य हैं जैसा उन्हें प्रोग्राम (निष्पादन, I/O, फाइल, संचार, त्रुटि) और उपयोगकर्ता (इंटरफ़ेस, उपयोगिताएँ) अनुभव करते हैं।
- कर्नेल हमेशा निवास करने वाला केंद्र है; मोनोलिथिक (सारी सेवाएँ अंदर, तेज़ — Linux) बनाम माइक्रोकर्नेल (न्यूनतम अंदर, विश्वसनीय — MINIX)।
- सिस्टम कॉल यूज़र-मोड → कर्नेल-मोड का द्वार है; पाँच प्रकार: प्रक्रिया नियंत्रण, फाइल, उपकरण, सूचना, संचार।
- UNIX/Linux: हार्डवेयर → कर्नेल → सिस्टम-कॉल इंटरफ़ेस → लाइब्रेरी/शेल/उपयोगिताएँ → उपयोगकर्ता; सिस्टम प्रोग्राम कर्नेल के बाहर OS का बदला जा सकने वाला टूलबॉक्स हैं।

अध्याय अभ्यास सेट

- OS के पाँच मुख्य कार्य गिनाइए और बताइए कि इस पाठ्यक्रम की कौन सी इकाई पहले चार में से हर एक का अध्ययन करती है।
- कर्नेल को परिभाषित कीजिए। मोनोलिथिक और माइक्रोकर्नेल डिज़ाइन की चार बिंदुओं पर तुलना कीजिए।
- सिस्टम कॉल क्या है? उसके पाँच चरण बनाइए/वर्णन कीजिए और उसके पाँच प्रकार हर एक के एक उदाहरण सहित गिनाइए।

- एक उपयोगकर्ता Linux पर `cp a.txt b.txt` टाइप करता है। पूरी कहानी बताइए: शेल, सिस्टम प्रोग्राम, सिस्टम कॉल, कर्नेल, हार्डवेयर।
- UNIX/Linux परतदार संरचना को पाँच लाइनों में समझाइए।
- सिस्टम प्रोग्राम बनाम एप्लिकेशन प्रोग्राम: दो अंतर, हर एक के दो उदाहरण।
- "C में `printf` और Python में `print` अंततः सिस्टम कॉल बन जाते हैं।" समझाइए कि लाइब्रेरी परत है ही क्यों।





अध्याय 3: प्रक्रिया अवधारणाएँ और प्रक्रियाओं पर कार्य

(BTEUP CSE तीसरा सेमेस्टर · विषय 3.1 · इकाई 2 — प्रक्रिया प्रबंधन)

आप क्या सीखेंगे

- प्रोग्राम बनाम प्रक्रिया — वह अंतर जिस पर पूरी इकाई खड़ी है
- प्रक्रिया की पाँच अवस्थाएँ और वह चित्र जो हर परीक्षक अपेक्षा करता है
- प्रोसेस कंट्रोल ब्लॉक (PCB) — OS हर प्रक्रिया के बारे में क्या रिकॉर्ड रखता है
- कार्य: प्रक्रियाओं को बनाना और समाप्त करना, प्रक्रिया वृक्ष, और कॉन्टेक्स्ट स्विच
- तीन शेड्यूलर और वे कतारें जिनमें प्रक्रियाएँ प्रतीक्षा करती हैं

3.1 प्रक्रिया क्या है?

प्रक्रिया निष्पादन में चल रहा प्रोग्राम है — मेमोरी में लोड की गई प्रोग्राम फाइल, साथ में वह सब कुछ जो उसे चलने के लिए चाहिए: उसकी वर्तमान निर्देश स्थिति, उसके चरों के मान, उसका स्टैक और वे संसाधन (फाइलें, मेमोरी) जो OS ने उसे दिए हैं। डिस्क पर रखा प्रोग्राम निष्क्रिय कोड है; प्रक्रिया वह कोड है जो जीवित होकर काम कर रहा है।

अंतर महसूस करना आसान है: एक प्रोग्राम कई प्रक्रियाएँ हो सकता है। एक ही ब्राउज़र प्रोग्राम को तीन विंडो में खोलिए और आपके पास एक प्रोग्राम लेकिन तीन प्रक्रियाएँ हैं — हर एक की अपनी मेमोरी, अपने पेज, अपनी अवस्था, कोई किसी को परेशान नहीं करती। आपके Python कार्य में भी यही हुआ था: Python इंटरप्रेटर एक प्रोग्राम है; आपकी चलाई हर स्क्रिप्ट उसकी एक अलग प्रक्रिया थी।

मेमोरी में प्रक्रिया के पहचाने जाने वाले भाग होते हैं: **टेक्स्ट** (कोड), **डेटा** (ग्लोबल चर), **हीप** (चलते समय आवंटित मेमोरी) और **स्टैक** (फंक्शन कॉल और स्थानीय डेटा) — साथ में CPU रजिस्टर और प्रोग्राम काउंटर (अगला निर्देश कौन सा है), जो तार्किक रूप से प्रक्रिया के साथ रहते हैं भले ही CPU कहीं और व्यस्त हो।

अभ्यास सेट 3.1

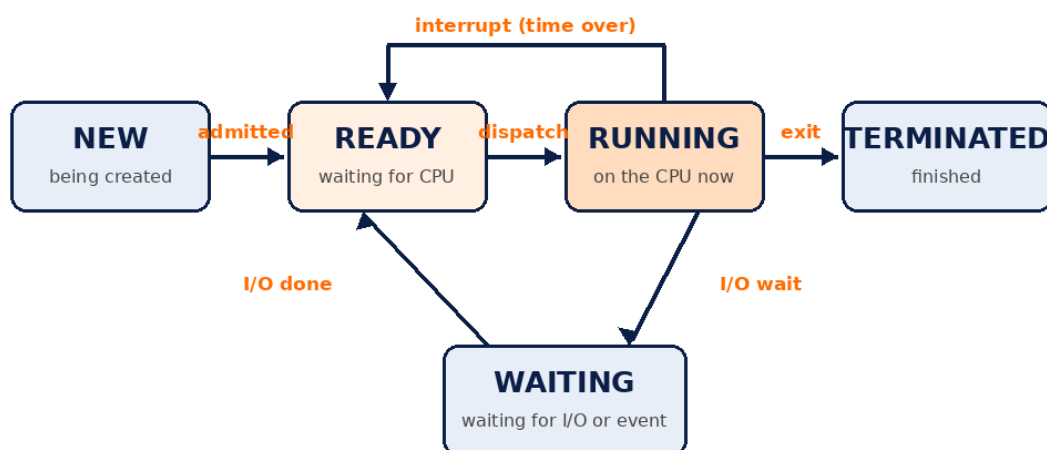
- एक प्रोग्राम पाँच बार खोला जाता है। कितने प्रोग्राम, कितनी प्रक्रियाएँ?
- जब फंक्शन फंक्शनों को बुलाते रहते हैं तो प्रक्रिया का कौन सा भाग बढ़ता है?
- "प्रक्रिया ही प्रोग्राम है" — सही है या नहीं? दो लाइनों में उत्तर दीजिए।

3.2 प्रक्रिया अवस्थाएँ

जैसे-जैसे प्रक्रिया जीती है, वह अवस्थाओं से गुज़रती है। पाँच मानक अवस्थाएँ:

- **New (नई)** — प्रक्रिया बनाई जा रही है (लोड हो रही है, उसे PCB और संसाधन मिल रहे हैं)।
- **Ready (तैयार)** — उसके पास CPU के अलावा सब कुछ है; वह ready कतार में अपनी बारी की प्रतीक्षा करती है।
- **Running (चल रही)** — उसके निर्देश CPU पर निष्पादित हो रहे हैं। एक CPU पर, किसी भी क्षण प्रति कोर केवल एक प्रक्रिया चल रही होती है।
- **Waiting (प्रतीक्षारत / blocked)** — वह तब तक आगे नहीं बढ़ सकती जब तक कोई घटना न हो जाए, आमतौर पर I/O का पूरा होना; खाली CPU भी अभी उसकी मदद नहीं कर सकता।
- **Terminated (समाप्त)** — वह समाप्त हो गई है; OS उसके संसाधन वापस ले लेता है।

The Process State Diagram



चित्र 3.1 - प्रक्रिया की पाँच अवस्थाएँ और उनके बीच के बदलाव।

तीरों को ध्यान से पढ़िए, परीक्षा का यही भाग है: प्रक्रिया **new** से **ready** में प्रवेश (**admitted**) पाती है; शेड्यूलर **ready** प्रक्रिया को **running** के लिए भेजता (**dispatch**) है; चलती प्रक्रिया **exit** से (समाप्त), **interrupt** से (उसका समय टुकड़ा समाप्त — वापस **ready**), या **I/O** प्रतीक्षा से (**waiting** में, **I/O** पूरा होने पर **ready** में वापस) निकलती है। ईमानदार ब्योरा नोट कीजिए: **I/O** पूरा होना प्रक्रिया को **सीधे running में नहीं, ready में** ले जाता है — उसे बाकी सब की तरह **dispatch** की प्रतीक्षा करनी पड़ती है।

अभ्यास सेट 3.2

- एक प्रक्रिया ने डिस्क से डेटा माँगा और डिस्क व्यस्त है। उसकी अवस्था, और क्रम में उसकी अगली दो अवस्थाएँ?
- क्या कोई प्रक्रिया **waiting** से सीधे **running** में जा सकती है? क्यों नहीं?
- 4-कोर CPU पर, एक क्षण में अधिकतम कितनी प्रक्रियाएँ **running** अवस्था में होती हैं?

3.3 प्रोसेस कंट्रोल ब्लॉक (PCB)

OS हर प्रक्रिया का रिकॉर्ड एक डेटा संरचना में रखता है जिसे **प्रोसेस कंट्रोल ब्लॉक (PCB)** कहते हैं — हर प्रक्रिया के लिए एक। इसकी आम सामग्री:

- **प्रोसेस ID (PID)** — प्रक्रिया की अद्वितीय संख्या।
- **प्रक्रिया अवस्था** — **new/ready/running/waiting/terminated**।

- **प्रोग्राम काउंटर** — निष्पादित होने वाले अगले निर्देश का पता।
- **CPU रजिस्टर** — सहेजे गए रजिस्टर मान, ठीक वहीं से फिर शुरू करने के लिए जहाँ वह रुकी थी।
- **CPU शेड्यूलिंग जानकारी** — प्राथमिकता, कतार में स्थिति।
- **मेमोरी प्रबंधन जानकारी** — प्रक्रिया को आवंटित मेमोरी, पेज टेबल और सीमाएँ।
- **लेखा जानकारी** — उपयोग किया गया CPU समय, समय सीमाएँ।
- **I/O स्थिति जानकारी** — उसके लिए वर्तमान में खुले उपकरण और फाइलें।

आपके OS Lab का Task Manager प्रयोग इसी खंड का दृश्य रूप है: ID, अवस्थाओं और CPU/मेमोरी आँकड़ों वाली प्रक्रिया सूची सिस्टम के PCB पर खुलती खिड़की है।

अभ्यास सेट 3.3

- एक प्रक्रिया काम के बीच में रुकती है और एक घंटे बाद ठीक उसी निर्देश से फिर शुरू होती है। PCB के कौन से दो क्षेत्र इसे संभव बनाते हैं?
- PCB खुली फाइलों की जानकारी रखता ही क्यों है?

3.4 प्रक्रियाओं पर कार्य

निर्माण। एक प्रक्रिया (parent / जनक) क्रिएट-प्रोसेस सिस्टम कॉल के द्वारा नई प्रक्रियाएँ (children / संतान) बनाती है — UNIX/Linux में `fork()`, जो जनक की लगभग-नकल बनाता है, जिसके बाद आमतौर पर एक और कॉल संतान में नया प्रोग्राम लोड करती है। संतानें अपनी संतानें बना सकती हैं, जिससे **प्रक्रिया वृक्ष** बनता है: Linux पर वृक्ष की जड़ पहली प्रक्रिया (`init/systemd`, PID 1) है, जिसे बूट के समय कर्नेल बनाता है; आपकी शुरू की हर प्रक्रिया उसके नीचे कहीं लटकी होती है। जब आप किसी ऐप पर डबल-क्लिक करते हैं, तो शेल/डेस्कटॉप प्रक्रिया जनक है और ऐप संतान है।

समाप्ति। प्रक्रिया काम पूरा होने पर `exit` सिस्टम कॉल से समाप्त होती है (सामान्य समाप्ति), या त्रुटि, सीमा से अधिक उपयोग या आवश्यकता न रहने पर OS या किसी अन्य अधिकृत प्रक्रिया द्वारा समाप्त की जाती है (असामान्य समाप्ति)। समाप्ति पर OS उसकी मेमोरी और संसाधन वापस लेता है और जनक को सूचित करता है; संतानें सिस्टम के अनुसार जड़ प्रक्रिया द्वारा गोद ली जा सकती हैं या जनक के साथ समाप्त की जा सकती हैं।

कॉन्टेक्स्ट स्विच। जब CPU एक प्रक्रिया से दूसरी पर जाता है, तो OS **कॉन्टेक्स्ट स्विच** करता है: वर्तमान प्रक्रिया की अवस्था (रजिस्टर, प्रोग्राम काउंटर) उसके PCB में सहेजना, और अगली प्रक्रिया की सहेजी अवस्था उसके PCB से लोड करना। इसमें समय लगता है जिसके दौरान कोई उपयोगी काम नहीं होता — शुद्ध अतिरिक्त भार (overhead) — यही कारण है कि स्विच केवल तब हर सेकंड हज़ारों बार होते हैं जब वे बहुत तेज़ हों, और समय टुकड़ा (अध्याय 4) बहुत छोटा नहीं होना चाहिए।

अभ्यास सेट 3.4

- इसके लिए प्रक्रिया वृक्ष बनाइए: सिस्टम प्रारंभ → लॉगिन शेल → शेल से शुरू किए गए एक संपादक और एक कंपाइलर।
- एक जनक समाप्त हो जाता है। इस खंड में बताई गई उसकी संतानों की दो संभावित गतियाँ बताइए।
- कॉन्टेक्स्ट स्विच समय को overhead क्यों कहते हैं? इसे दुर्लभ बनाना किस चीज़ से संतुलित होता है?

3.5 शेड्यूलर और कतारें (पहली झलक)

प्रक्रियाएँ **कतारों** में प्रतीक्षा करती हैं: ready कतार (तैयार प्रक्रियाएँ, मेमोरी में), और उपकरण कतारें (किसी खास उपकरण की प्रतीक्षा करती प्रक्रियाएँ)। तीन शेड्यूलर प्रक्रियाओं को स्थानों के बीच ले जाते हैं:

- **दीर्घकालिक शेड्यूलर (Long-term / job scheduler)** — चुनता है कि कौन से नए काम मेमोरी में प्रवेश पाएँगे (डिस्क पर रखे काम संग्रह से) और मल्टीप्रोग्रामिंग की मात्रा (मेमोरी में कितनी प्रक्रियाएँ हैं) नियंत्रित करता है। दुर्लभ रूप से चलता है — सेकंडों या मिनटों के अंतर पर।
- **अल्पकालिक शेड्यूलर (Short-term / CPU scheduler)** — चुनता है कि कौन सी ready प्रक्रिया को अगला CPU मिलेगा। बहुत अक्सर चलता है — लगभग हर घटना के बाद; अध्याय 4 पूरी तरह उसके एल्गोरिदम के बारे में है।
- **मध्यकालिक शेड्यूलर (Medium-term scheduler)** — मेमोरी भीड़भाड़ वाली होने पर किसी प्रक्रिया को अस्थायी रूप से मेमोरी से डिस्क पर हटा देता है (swap out) और बाद में वापस लाता है (यह अध्याय 8 के swapping से जुड़ा है)।

अभ्यास सेट 3.5

- कौन सा शेड्यूलर मल्टीप्रोग्रामिंग की मात्रा तय करता है? कौन सा सबसे तेज़ होना चाहिए, और क्यों?
- मेमोरी खाली करने के लिए एक प्रक्रिया डिस्क पर swap out की जाती है। यह किस शेड्यूलर ने किया, और वह प्रतीक्षा कहाँ करती है?

आम गलतियाँ

- प्रोग्राम और प्रक्रिया को पर्याय लिखना। प्रोग्राम निष्क्रिय है, डिस्क पर; प्रक्रिया प्रोग्राम का जीवित रूप है — एक प्रोग्राम, कई प्रक्रियाएँ।
- I/O पूर्ति को waiting → running बनाना। यह waiting → ready है।
- कॉन्टेक्स्ट स्विच को "मुफ्त" कहना। यह शुद्ध overhead है; इसकी लागत शेड्यूलिंग विकल्पों के मायने रखने का एक कारण है।
- दीर्घकालिक और अल्पकालिक शेड्यूलर को मिलाना: मेमोरी में प्रवेश (दुर्लभ) बनाम CPU हस्तांतरण (लगातार)।

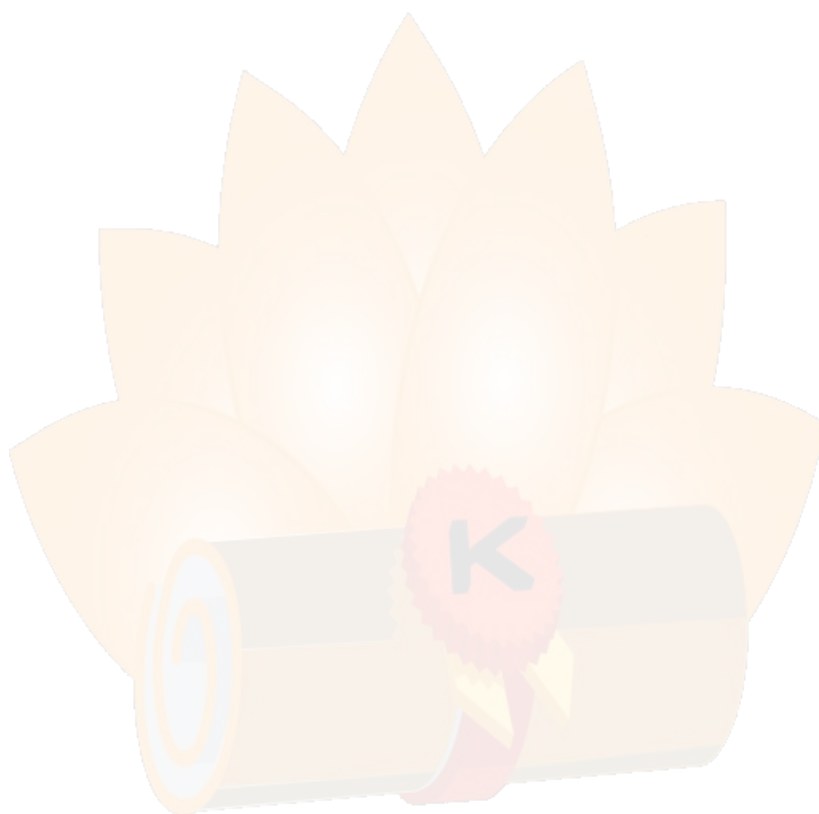
सारांश

- प्रक्रिया निष्पादन में चल रहा प्रोग्राम है, जिसकी अपनी मेमोरी (टेक्स्ट, डेटा, हीप, स्टैक) और अवस्था होती है।
- पाँच अवस्थाएँ: new, ready, running, waiting, terminated — बदलाव प्रवेश, dispatch, interrupt, I/O प्रतीक्षा/पूर्ति और exit से होते हैं।
- PCB प्रक्रिया के बारे में सब कुछ रिकॉर्ड करता है: PID, अवस्था, प्रोग्राम काउंटर, रजिस्टर, शेड्यूलिंग, मेमोरी, लेखा, I/O।
- जनक संतानें बनाते हैं (PID 1 जिसकी जड़ है, वह प्रक्रिया वृक्ष); समाप्ति संसाधन वापस लेती है; कॉन्टेक्स्ट स्विच एक PCB सहेजता और दूसरा लोड करता है, overhead लागत पर।
- दीर्घकालिक (प्रवेश), अल्पकालिक (CPU), मध्यकालिक (swapping) शेड्यूलर प्रक्रियाओं को मेमोरी, ready कतार और उपकरणों के बीच ले जाते हैं।

अध्याय अभ्यास सेट

- प्रक्रिया को परिभाषित कीजिए। यह प्रोग्राम से कैसे भिन्न है? तीन-विंडो वाला उदाहरण अपने शब्दों में दीजिए।
- प्रक्रिया अवस्था चित्र बनाइए और हर तीर समझाइए।
- PCB की सामग्री गिनाइए और उसमें प्रोग्राम काउंटर की भूमिका बताइए।
- कॉन्टेक्स्ट स्विच क्या है? उसके दो आधे भागों का वर्णन कीजिए और बताइए कि यह overhead क्यों है।

- प्रक्रिया निर्माण और प्रक्रिया वृक्ष समझाइए। PID 1 में क्या खास है?
- तीनों शेड्यूलरों की तुलना कीजिए: हर एक क्या तय करता है, और हर एक कितनी बार चलता है।
- OS Lab में आप Task Manager देखेंगे — ऐसे चार PCB क्षेत्र बताइए जो वहाँ प्रतिबिंबित दिखते हैं।





अध्याय 4: CPU शेड्यूलिंग

(BTEUP CSE तीसरा सेमेस्टर · विषय 3.1 · इकाई 2 — प्रक्रिया प्रबंधन)

आप क्या सीखेंगे

- CPU शेड्यूलिंग क्या तय करती है, और उसे मापने के पैमाने (मानदंड)
- प्रीएम्प्टिव बनाम नॉन-प्रीएम्प्टिव शेड्यूलिंग
- FCFS, SJF, SRTF, Round Robin और Priority — हर एक पूरी तरह हल किए गए Gantt चार्ट उदाहरण के साथ
- काफ़िला प्रभाव (convoy effect), भुखमरी (starvation), और एजिंग (aging)

4.1 शेड्यूलिंग क्या है, और इसे कैसे मापा जाता है

जब भी एक से अधिक प्रक्रिया तैयार होती है, किसी को चुनना पड़ता है कि अगला CPU किसे मिलेगा। यही चुनाव — अल्पकालिक शेड्यूलर (अध्याय 3) द्वारा किया गया — **CPU शेड्यूलिंग** है। अलग-अलग चुनाव बहुत अलग अनुभव देते हैं, इसलिए शेड्यूलिंग एल्गोरिदम इन मानदंडों से मापे जाते हैं:

- **CPU उपयोग (utilisation)** — समय का वह प्रतिशत जब CPU उपयोगी काम में व्यस्त रहता है। अधिक बेहतर।
- **थ्रूपुट (Throughput)** — प्रति इकाई समय में पूरी हुई प्रक्रियाओं की संख्या। अधिक बेहतर।
- **टर्नअराउंड समय (TAT)** — प्रक्रिया के आगमन से उसके पूरा होने तक: $TAT = \text{पूरा होने का समय} - \text{आगमन समय}$ । कम बेहतर।
- **प्रतीक्षा समय (WT)** — ready कतार में प्रतीक्षा में बिताया कुल समय: $WT = TAT - \text{Burst समय}$ । कम बेहतर।
- **प्रत्युत्तर समय (Response time)** — आगमन से पहली बार CPU मिलने तक (बातचीत वाले उपयोग के लिए मायने रखता है)।

दो कार्यशैलियाँ: **नॉन-प्रीएम्प्टिव** शेड्यूलिंग में, प्रक्रिया को CPU मिलने के बाद वह उसे पूरा होने या I/O के लिए रुकने तक रखती है — सरल, लेकिन एक लंबी प्रक्रिया सबको रोक सकती है। **प्रीएम्प्टिव** शेड्यूलिंग में, OS CPU छीन सकता है (समय टुकड़ा समाप्त, या अधिक ज़रूरी प्रक्रिया का आगमन) — अधिक न्यायसंगत और उत्तरदायी, अधिक कॉन्टेक्स्ट स्विच की लागत पर। नीचे हर उदाहरण बताता है कि वह किस शैली का उपयोग करता है।

सभी हल उदाहरणों की विधि: Gantt चार्ट बनाइए (वह पट्टी जो दिखाती है कि कब कौन सी प्रक्रिया चलती है), हर पूरा होने का समय पढ़िए, फिर $TAT = CT - AT$ और $WT = TAT - BT$ लगाइए। Burst समय (BT) प्रक्रिया को चाहिए CPU समय है; आगमन समय (AT) वह है जब वह ready हुई।

अभ्यास सेट 4.1

- एक प्रक्रिया 2 पर आती है, 11 पर पूरी होती है, और उसे 6 इकाई CPU चाहिए। उसका TAT और WT?
- बातचीत करने वाला उपयोगकर्ता किस मानदंड को सबसे सीधे महसूस करता है? सर्वर मालिक किस पर नज़र रखता है?

4.2 FCFS — पहले आओ, पहले पाओ

प्रक्रियाएँ आगमन क्रम में चलती हैं, नॉन-प्रीएम्प्टिव रूप से — एक न्यायसंगत कतार, टिकट काउंटर जैसी।

हल उदाहरण 1. तीनों समय 0 पर आती हैं, क्रम P1, P2, P3 में। Burst समय: $P1 = 24$, $P2 = 3$, $P3 = 3$ ।

Gantt चार्ट: | P1 (0–24) | P2 (24–27) | P3 (27–30) |

प्रक्रिया	BT	CT	$TAT = CT - AT$	$WT = TAT - BT$
P1	24	24	0	0
P2	3	27	24	21
P3	3	30	27	24

औसत $WT = (0 + 24 + 27) / 3 = 17$ । औसत $TAT = (24 + 27 + 30) / 3 = 27$ ।

काफ़िला प्रभाव (convoy effect): P2 और P3 — छोटे 3-इकाई काम — 24 और 27 इकाई प्रतीक्षा करते रहे क्योंकि एक लंबा काम पहले आ गया, ठीक जैसे एक धीमा ट्रक कारों की लाइन को रोक देता है। क्रम P2, P3, P1 होता तो औसत प्रतीक्षा समय 3 हो जाता (जाँचिए)। FCFS सरल है और

भुखमरी से मुक्त, लेकिन औसत प्रतीक्षा समय आगमन क्रम पर बहुत निर्भर करता है और आमतौर पर खराब रहता है। आपका OS Lab इसी कारण पहले FCFS का अनुकरण करता है: यह वह आधाररेखा है जिसे हर दूसरे एल्गोरिदम को हराना होता है।

अभ्यास सेट 4.2

- ऊपर के दावे की जाँच कीजिए: क्रम P2, P3, P1 औसत WT = 3 देता है।
- प्रक्रियाएँ (AT, BT): A(0, 5), B(2, 2), C(3, 1)। FCFS औसत WT? (आगमनों का भी ध्यान रखिए।)

4.3 SJF और SRTF — सबसे छोटा काम पहले

SJF (Shortest Job First): सबसे कम burst समय वाली ready प्रक्रिया अगली चलती है।
नॉन-प्रीएम्प्टिव रूप: चुनाव तब होता है जब CPU खाली होता है।

हल उदाहरण 1 (जारी)। वही तीन प्रक्रियाएँ (24, 3, 3), SJF क्रम है P2, P3, P1।

Gantt चार्ट: | P2 (0-3) | P3 (3-6) | P1 (6-30) |

WT: P2 = 0, P3 = 3, P1 = 6। औसत WT = 3 — एक ही काम पर FCFS के 17 के मुकाबले। SJF प्रक्रियाओं के किसी दिए समूह के लिए संभव सबसे अच्छा औसत प्रतीक्षा समय देता है — यह सिद्ध है, इसीलिए यह वह मानक है जिससे दूसरे एल्गोरिदम तुलना किए जाते हैं। इसकी कमज़ोरी व्यावहारिक है: OS किसी प्रक्रिया का ठीक भविष्य burst समय नहीं जान सकता; वह केवल पिछले व्यवहार से अनुमान लगा सकता है। दूसरी कमज़ोरी: लंबी प्रक्रियाएँ नए आते छोटे कामों से हारती रह सकती हैं — भुखमरी, जिसका इलाज खंड 4.5 में है।

SRTF (Shortest Remaining Time First): SJF का प्रीएम्प्टिव रूप — जब भी कोई प्रक्रिया आती है, यदि उसका burst चलती प्रक्रिया के शेष समय से कम हो, तो वह अधिकार ले लेती है।

हल उदाहरण 2. P1 (AT 0, BT 8), P2 (AT 1, BT 4), P3 (AT 2, BT 9), P4 (AT 3, BT 5), SRTF से शेड्यूल किए गए।

चरण दर चरण: 0 पर केवल P1 है, वह चलती है; 1 पर P2 4 के साथ आती है < P1 का शेष 7, इसलिए P2 अधिकार लेती है; P3 (9) और P4 (5) P2 के चलने के दौरान आते हैं लेकिन कोई P2 के शेष समय को नहीं हराता, इसलिए P2 समय 5 पर पूरी होती है। शेष: P4 (5) सबसे छोटा है, 5-10 चलता है; फिर P1 (7), 10-17; फिर P3, 17-26।

Gantt चार्ट: | P1 (0-1) | P2 (1-5) | P4 (5-10) | P1 (10-17) | P3 (17-26) |

प्रक्रिया	AT	BT	CT	TAT	WT	...	...	...	...	...	...	P1	0	8	17	17					
9		P2	1	4	5	4	0		P3	2	9	26	24	15		P4	3	5	10	7	2

औसत $WT = (9 + 0 + 15 + 2) / 4 = 6.5$ । औसत $TAT = (17 + 4 + 24 + 7) / 4 = 13$ ।

तुलना के लिए, उसी समूह पर नॉन-प्रीएम्प्टिव SJF औसत WT 7.75 देता है (अभ्यास सेट में जाँचिए) — प्रीएम्पशन ने यहाँ छोटा सुधार खरीदा, अतिरिक्त कॉन्टेक्स्ट स्विच की कीमत पर।

अभ्यास सेट 4.3

- हल उदाहरण 2 के लिए SJF (नॉन-प्रीएम्प्टिव) आँकड़ों की जाँच कीजिए: क्रम, पूरा होने के समय, औसत WT 7.75।
- असली OS पूर्ण SJF लागू क्यों नहीं कर सकता? वह इसके बदले क्या करता है (एक लाइन)?

4.4 Round Robin — बारी-बारी

Round Robin (RR): हर ready प्रक्रिया को एक **समय क्वांटम** (समय टुकड़े) के लिए CPU मिलता है, फिर अधूरी रहने पर वह कतार के पीछे जाती है; डिज़ाइन से प्रीएम्प्टिव। यह FCFS + एक टाइमर है — टाइम-शेयरिंग के लिए बना, जहाँ किसी एक काम को जल्दी पूरा करने से अधिक मायने हर एक को तुरंत प्रत्युत्तर मिलना रखता है।

हल उदाहरण 3. उदाहरण 1 की वही तीन प्रक्रियाएँ ($P1 = 24$, $P2 = 3$, $P3 = 3$, सभी 0 पर), क्वांटम = 4।

कतार की कहानी: $P1$ 0-4 चलती है (20 शेष, पीछे जाती है); $P2$ 4-7 चलकर पूरी होती है (उसे केवल 3 चाहिए थे); $P3$ 7-10 चलकर पूरी होती है; $P1$ फिर अकेली 30 तक चलती है।

Gantt चार्ट: | $P1$ (0-4) | $P2$ (4-7) | $P3$ (7-10) | $P1$ (10-30) |

WT : $P1 = 30 - 24 = 6$; $P2 = 7 - 3 = 4$; $P3 = 10 - 3 = 7$ । औसत $WT = 17/3 \approx 5.67$ ।

औसत $TAT = (30 + 7 + 10)/3 \approx 15.67$ ।

ऑकड़ों में समझौता पढ़िए: RR की औसत प्रतीक्षा (5.67) SJF (3) और FCFS (17) के बीच है, लेकिन हर प्रक्रिया को 4 इकाइयों के भीतर CPU मिला — प्रत्युत्तर समय वह जगह है जहाँ RR जीतता है, इसीलिए बातचीत वाले सिस्टम इसका उपयोग करते हैं। क्वांटम समायोजन का घुंडी है: बहुत बड़ा तो RR FCFS में बदल जाता है; बहुत छोटा तो कॉन्टेक्स्ट-स्विच overhead मशीन को खा जाता है (अध्याय 3)। आम अनुभव-नियम: क्वांटम ऐसा चुनिए कि अधिकांश प्रक्रियाएँ एक टुकड़े में पूरी हो जाएँ।

अभ्यास सेट 4.4

- उदाहरण 3 को क्वांटम = 2 के साथ फिर कीजिए और औसत प्रतीक्षा समय व कॉन्टेक्स्ट स्विचों की संख्या की तुलना कीजिए।
- क्वांटम = 30 के साथ, इस उदाहरण के लिए RR क्या बन जाता है? सामान्य नियम बताइए।

4.5 प्रायोरिटी शेड्यूलिंग, भुखमरी और एजिंग

हर प्रक्रिया के पास एक **प्राथमिकता (priority)** संख्या होती है; CPU सबसे उच्च प्राथमिकता वाली ready प्रक्रिया को जाता है (इस किताब में छोटी संख्या का अर्थ उच्च प्राथमिकता है — परीक्षा में अपना नियम लिखिए, दोनों मौजूद हैं)। यह प्रीएम्प्टिव (नई आई उच्च-प्राथमिकता प्रक्रिया अधिकार ले लेती है) या नॉन-प्रीएम्प्टिव हो सकता है।

हल उदाहरण 4 (नॉन-प्रीएम्प्टिव)। उदाहरण 2 का वही समूह, प्राथमिकताओं सहित: P1 (AT 0, BT 8, प्राथमिकता 3), P2 (AT 1, BT 4, प्राथमिकता 2), P3 (AT 2, BT 9, प्राथमिकता 4), P4 (AT 3, BT 5, प्राथमिकता 1)।

0 पर केवल P1 मौजूद है, इसलिए वह 8 पर पूरी होने तक चलती है (नॉन-प्रीएम्प्टिव — प्राथमिकता बीच में नहीं रोक सकती)। फिर ready प्राथमिकताएँ हैं P4 (1), P2 (2), P3 (4): P4 8–13 चलती है, P2 13–17, P3 17–26।

Gantt चार्ट: | P1 (0–8) | P4 (8–13) | P2 (13–17) | P3 (17–26) |

WT: P1 = 0, P2 = 17 – 1 – 4 = 12, P3 = 26 – 2 – 9 = 15, P4 = 13 – 3 – 5 = 5। औसत WT = $32/4 = 8$ ।

भुखमरी (Starvation): कम-प्राथमिकता प्रक्रिया अनिश्चित काल तक प्रतीक्षा कर सकती है यदि उच्च-प्राथमिकता काम आता रहे — उसकी प्रतीक्षा असीमित बढ़ती है, जबकि सिस्टम व्यस्त और "सही" रहता है। मानक इलाज **एजिंग (aging)** है: प्रतीक्षारत प्रक्रिया की प्राथमिकता समय के साथ धीरे-धीरे सुधरती है, इसलिए अंततः वह नए आने वालों से आगे निकलकर चलती है। SJF को भी यही रोग दूसरे रूप

में है (लंबे काम भूखे रहते हैं) और वहाँ भी इलाज एजिंग ही है। प्रायोरिटी शेड्यूलिंग वह तरीका है जिससे असली सिस्टम कहते हैं "यह काम अधिक मायने रखता है" — सिस्टम प्रक्रियाएँ पृष्ठभूमि कामों से ऊपर, आपके टैप पृष्ठभूमि सिंक से ऊपर।

अभ्यास सेट 4.5

- उदाहरण 4 को प्रीएम्प्टिव बनाइए और फिर गिनिए: समय 1 पर कौन चलती है, और नए पूरा होने के समय क्या हैं?
- एक सर्वर एजिंग के बिना प्रायोरिटी शेड्यूलिंग चलाता है और एक बैकअप काम एक सप्ताह से नहीं चला, हालाँकि CPU कभी खाली नहीं रहा। समझाइए, और एक लाइन में ठीक कीजिए।

4.6 पाँचों एल्गोरिदम अगल-बगल

| एल्गोरिदम | नियम | प्रीएम्प्टिव? | शक्ति | कमज़ोरी | |---|---|---|---|---| | FCFS | आगमन क्रम | नहीं | सरल, भुखमरी नहीं | काफ़िला प्रभाव, खराब औसत WT | | SJF | सबसे छोटा burst अगला | नहीं (SRTF है) | सर्वश्रेष्ठ औसत WT (सिद्ध) | भविष्य ज्ञान चाहिए; लंबे काम भूखे | | SRTF | सबसे कम शेष समय | हाँ | आगमनों के साथ सर्वश्रेष्ठ औसत WT | कई कॉन्टेक्स्ट स्विच; भुखमरी | | Round Robin | तय समय क्वांटम बारी से | हाँ | सर्वश्रेष्ठ प्रत्युत्तर समय, न्यायसंगत | क्वांटम छोटा हो तो overhead; औसत WT मध्यम | | Priority | सर्वोच्च प्राथमिकता पहले | कोई भी | ज़रूरी काम पहले | एजिंग बिना भुखमरी |

आम गलतियाँ

- WT को सूत्रों के बजाय Gantt चार्ट की स्थितियों से सीधे गिनना; आगमन मौजूद हों तो चार्ट विधि भटकाती है — हमेशा सूत्र इस्तेमाल कीजिए।
- SRTF में, नए आने वाले के burst की तुलना चलती प्रक्रिया के मूल burst से करना, उसके शेष समय के बजाय।
- Round Robin में, यह भूलना कि अपने क्वांटम के अंदर पूरी होने वाली प्रक्रिया तुरंत निकल जाती है (उदाहरण 3 में P2 7 पर पूरी हुई, 8 पर नहीं), या अधूरी प्रक्रिया को कतार के पीछे भेजना भूलना।
- प्राथमिकता उदाहरण बिना यह बताए शुरू करना कि छोटी या बड़ी संख्या का अर्थ उच्च प्राथमिकता है।

सारांश

- शेड्यूलिंग मानदंड: CPU उपयोग, थ्रूपुट, टर्नअराउंड समय, प्रतीक्षा समय, प्रत्युत्तर समय; $TAT = CT - AT$, $WT = TAT - BT$ ।
- FCFS: आगमन क्रम; सरल, भुखमरी-मुक्त, काफ़िला प्रभाव से पीड़ित। उदाहरण: औसत WT 17।
- SJF: सबसे छोटा burst पहले; सिद्ध रूप से सर्वश्रेष्ठ औसत WT (उदाहरण में 3) लेकिन अनुमानित burst चाहिए; SRTF इसका प्रीएम्प्टिव रूप है (उदाहरण 2 में औसत WT 6.5)।
- Round Robin: क्वांटम-आधारित बारी; निष्पक्षता और प्रत्युत्तर समय, क्वांटम का आकार समझौता है (उदाहरण 3 में औसत $WT \approx 5.67$)।
- Priority: सर्वोच्च प्राथमिकता पहले; भुखमरी का इलाज एजिंग (उदाहरण 4 औसत WT 8)।

अध्याय अभ्यास सेट

- प्रक्रियाएँ (AT, BT): A(0, 6), B(1, 3), C(2, 2), D(4, 4)। औसत WT और TAT गिनिए (a) FCFS
(b) SJF नॉन-प्रीएम्प्टिव
(c) SRTF के अंतर्गत।
- पिछले प्रश्न को Round Robin, क्वांटम 2 के साथ फिर कीजिए।
- उदाहरण 2 के लिए प्राथमिकताओं $P1=2$, $P2=4$, $P3=1$, $P4=3$ के साथ नॉन-प्रीएम्प्टिव priority का Gantt चार्ट बनाइए और औसत WT गिनिए।
- पाँचों शेड्यूलिंग मानदंडों को एक-एक लाइन में परिभाषित कीजिए।
- ट्रक-और-कारों वाली तस्वीर के साथ काफ़िला प्रभाव समझाइए, और उस एल्गोरिदम का नाम बताइए जो इससे पीड़ित है।
- "SJF सर्वोत्तम है फिर भी पूर्ण रूप में लागू नहीं हो सकता।" चार लाइनों में विरोधाभास सुलझाइए।
- एजिंग क्या है? इस अध्याय के किन दो एल्गोरिदम को इसकी ज़रूरत है, और क्यों?
(आंकिक अभ्यास प्रश्नों के उत्तर प्रश्न बैंक, इकाई 2 में हल किए गए हैं।)



अध्याय 5: अंतर-प्रक्रिया संचार, सिंक्रनाइज़ेशन और क्रिटिकल सेक्शन

(BTEUP CSE तीसरा सेमेस्टर · विषय 3.1 · इकाई 2 — प्रक्रिया प्रबंधन)

आप क्या सीखेंगे

- प्रक्रियाओं को संवाद क्यों चाहिए (IPC) और दो मॉडल: साझा मेमोरी और संदेश भेजना
- रेस कंडीशन — सही प्रोग्राम मिलकर गलत परिणाम कैसे देते हैं
- क्रिटिकल सेक्शन और किसी भी सही हल की तीन शर्तें
- सेमाफ़ोर: क्रिटिकल सेक्शन की रखवाली का मानक औज़ार
- क्लासिक समस्याएँ: उत्पादक-उपभोक्ता, पाठक-लेखक, डाइनिंग फ़िलॉसफ़र्स

5.1 अंतर-प्रक्रिया संचार (IPC)

प्रक्रियाएँ मूल रूप से स्वतंत्र होती हैं — अलग मेमोरी, अलग संसार (अध्याय 3)। लेकिन असली काम अक्सर सहयोगी काम होता है: एक प्रक्रिया डेटा बनाती है जिसे दूसरा उपयोग करता है, कई प्रक्रियाएँ एक प्रिंटर कतार साझा करती हैं, ब्राउज़र के टैब एक डाउनलोड मैनेजर साझा करते हैं। **अंतर-प्रक्रिया संचार (IPC)** तंत्रों का वह समूह है जो OS प्रक्रियाओं को डेटा का आदान-प्रदान करने और अपने कार्यों में समन्वय करने के लिए देता है। प्रक्रियाएँ चार क्लासिक कारणों से संवाद करती हैं: **सूचना साझा करना** (कई प्रक्रियाओं को एक ही डेटा चाहिए), **गणना गति बढ़ाना** (काम कई CPU या मशीनों में बाँटा), **मॉड्यूलरिटी** (सहयोगी भागों से बना सिस्टम), और **सुविधा** (कई जुड़े काम एक साथ करना)।

OS दो बुनियादी मॉडल देता है:

- **साझा मेमोरी (Shared memory)** — OS मेमोरी का एक क्षेत्र कई प्रक्रियाओं के एड्रेस स्पेस में जोड़ देता है; वे उस क्षेत्र को पढ़-लिखकर संवाद करती हैं, जैसे सहकर्मी एक ही साझा व्हाइटबोर्ड पर लिखते हैं। तेज़ (सेटअप के बाद, हर संदेश में कर्नेल शामिल नहीं), लेकिन प्रक्रियाओं को खुद ध्यान रखना होगा कि वे टकराएँ नहीं — जो ठीक अगले खंड की समस्या है।

- **संदेश भेजना (Message passing)** — प्रक्रियाएँ OS के द्वारा संदेश भेजती और पाती हैं (send / receive कार्य; pipe Linux शेल का रोज़मर्रा उदाहरण है, जहाँ एक कमांड का आउटपुट अगले में बहता है)। हर संदेश धीमा (हर आदान-प्रदान कर्नेल को पार करता है), लेकिन बनावट से सुरक्षित: लड़ने के लिए कोई साझा क्षेत्र ही नहीं है।

| आधार | साझा मेमोरी | संदेश भेजना | |---|---|---| | डेटा कैसे चलता है | एक साझा क्षेत्र पढ़ना/लिखना | OS के द्वारा send() और receive() | | गति | तेज़ | धीमा (हर बार कर्नेल शामिल) | | सुरक्षा | प्रक्रियाओं को खुद सिंक्रनाइज़ करना होगा | साझा डेटा नहीं, कम टकराव | | सर्वोत्तम उपयोग | बढ़ा या बार-बार डेटा आदान-प्रदान | छोटे संदेश, अलग मशीनें |

अभ्यास सेट 5.1

- एक वीडियो संपादक के मॉड्यूल हर सेकंड कई बार पूरे फ्रेम बदलते हैं; दो दफ़्तरी PC रोज़ की रिपोर्ट बदलते हैं। हर एक के लिए मॉडल चुनिए और एक लाइन में कारण दीजिए।
- दो शेल कमांडों के बीच pipe को साझा मेमोरी नहीं, संदेश भेजना क्यों माना जाता है?

5.2 सिंक्रनाइज़ेशन क्यों: रेस कंडीशन

साझा डेटा उपयोग करने वाली सहयोगी प्रक्रियाएँ उसे बिगाड़ सकती हैं — खराब कोड से नहीं, खराब समय से। एक साझा काउंटर लीजिए जिसे दो प्रक्रियाएँ एक-एक बार बढ़ाती हैं, 4 से शुरू होकर। "बढ़ाना" असल में तीन चरण हैं: मान पढ़ो, 1 जोड़ो, वापस लिखो। अब अंतर्गुथन (interleaving) कीजिए:

प्रक्रिया A 4 पढ़ती है → प्रक्रिया B 4 पढ़ती है → A 5 लिखती है → B 5 लिखती है। दो बार बढ़ाया गया; काउंटर 6 होना चाहिए था; वह 5 है। अंतर्गुथन अलग ढंग से चलाइए तो वह 4 या 5 या 6 पर समाप्त हो सकता है — परिणाम दौड़ (race) पर निर्भर करता है।

रेस कंडीशन ठीक यही है: वह स्थिति जहाँ कई प्रक्रियाएँ एक ही डेटा को एक ही समय में एक्सेस और बदलती हैं और अंतिम परिणाम एक्सेस के क्रम पर निर्भर करता है — इसलिए वह गलत हो सकता है। संयुक्त बैंक खाता दाँव दिखाता है: परिवार के दो सदस्य एक ही क्षण दो ATM से निकालते हैं; दोनों मशीनें एक ही शेष पढ़ती हैं, दोनों घटाती हैं, एक निकासी कभी दर्ज नहीं होती। इलाज **प्रोसेस सिंक्रनाइज़ेशन** है: एक अनुशासन जो प्रक्रियाओं को खतरनाक क्षेत्र में साझा डेटा का उपयोग एक-एक करके करने देता है, ताकि परिणाम समय पर निर्भर होना बंद कर दें।

अभ्यास सेट 5.2

- काउंटर को 10 से शुरू करके, ऐसा अंतर्गुथन बनाइए जहाँ तीन बढ़तें (A, B, C की एक-एक) काउंटर को 11 पर समाप्त करें।

- "हर प्रोग्राम अलग-अलग सही था, इसलिए सिस्टम सही होना चाहिए।" दो लाइनों में खंडन कीजिए।

5.3 क्रिटिकल सेक्शन और उसकी तीन शर्तें

प्रक्रिया के कोड का वह भाग जो साझा डेटा को छूता है — काउंटर, खाता, साझा बफ़र — उसका **क्रिटिकल सेक्शन** है। पूरी समस्या एक नियम में सिमटती है: किसी भी क्षण अधिक से अधिक एक प्रक्रिया अपने क्रिटिकल सेक्शन के अंदर (एक ही साझा डेटा के लिए) हो सकती है। सही हल को तीन शर्तें पूरी करनी चाहिए, और परीक्षा के उत्तरों में तीनों नाम आने चाहिए:

- **परस्पर अपवर्जन (Mutual exclusion)** — यदि एक प्रक्रिया अपने क्रिटिकल सेक्शन में है, तो कोई अन्य अपना (उसी डेटा के लिए) प्रवेश नहीं कर सकती।
- **प्रगति (Progress)** — यदि क्रिटिकल सेक्शन खाली है और कुछ प्रक्रियाएँ प्रवेश चाहती हैं, तो अगला कौन प्रवेश करेगा इसका चुनाव हमेशा के लिए नहीं टाला जा सकता, और अपने क्रिटिकल सेक्शन से बाहर की प्रक्रियाएँ चुनाव को नहीं रोक सकतीं।
- **सीमित प्रतीक्षा (Bounded waiting)** — किसी प्रक्रिया के प्रवेश माँगने के बाद, उसके पहले दूसरों के प्रवेश की एक सीमा होती है; कोई हमेशा प्रतीक्षा नहीं करता (भुखमरी नहीं)।

एक-लेन वाले पुल की तस्वीर लीजिए: परस्पर अपवर्जन = एक समय में पुल पर एक वाहन (यहाँ एक ही दिशा को एक उपयोगकर्ता गिनना); प्रगति = पुल खाली हो और कारें प्रतीक्षा करें तो किसी को तुरंत आगे बढ़ाया जाए; सीमित प्रतीक्षा = कोई चालक हमेशा कार पर कार गुज़रती न देखता रहे जबकि वह प्रतीक्षा कर रहा हो।

अभ्यास सेट 5.3

- एक हल परस्पर अपवर्जन की गारंटी देता है लेकिन एक तेज़ प्रक्रिया को बार-बार प्रवेश करने देता है जबकि दूसरी प्रतीक्षा करती है। कौन सी शर्त टूटती है?
- पुल वाली तस्वीर में, कौन सी शर्त टूटती है यदि परिचारक चाय के लिए चला जाए जबकि दोनों ओर कारों की कतार हो और पुल खाली खड़ा हो?

5.4 सेमाफ़ोर — मानक रक्षक

सेमाफ़ोर OS द्वारा रखा गया एक पूर्णांक चर है जिसे प्रक्रियाएँ साझा संसाधनों तक पहुँच में समन्वय के लिए उपयोग करती हैं: इसे केवल दो अविभाज्य (atomic) क्रियाओं से छुआ जाता है — **wait** (जिसे P या acquire भी कहते हैं), जो मान घटाती है और मान ऋणात्मक होने पर बुलाने वाले को रोक देती है, और **signal** (जिसे V या release भी कहते हैं), जो मान बढ़ाती है और कोई प्रतीक्षारत प्रक्रिया हो तो

उसे जगाती है। क्योंकि wait और signal atomic हैं — कोई अन्य प्रक्रिया किसी एक के बीच में नहीं घुस सकती — सेमाफ़ोर एक-एक-करके वाला नियम लागू कर सकता है जो कच्ची साझा मेमोरी नहीं कर सकती।

दो प्रकार:

- **बाइनरी सेमाफ़ोर (mutex)** — मान केवल 0 या 1। क्रिटिकल सेक्शन के चारों ओर लपेटा गया, यह ठीक पुल का द्वार है: प्रवेश से पहले wait, निकलते समय signal; दूसरी प्रक्रिया 0 पाती है और प्रतीक्षा करती है।
- **काउंटिंग सेमाफ़ोर** — मान समान संसाधनों की गिनती करता है: 3 से शुरू होकर, यह तीन प्रक्रियाओं को तीनों प्रिंटरों/डेटाबेस कनेक्शनों का एक साथ उपयोग करने देता है और चौथी को तब तक रोकता है जब तक कोई signal न करे।

प्रतीक्षा शैली पर एक नोट: सबसे सरल सेमाफ़ोर रुकी प्रक्रिया को busy-wait (घूमते रहना, बार-बार जाँचना) कराता है — बहु-कोर सिस्टमों पर बहुत छोटी प्रतीक्षाओं के लिए स्वीकार्य, अन्यथा बर्बादी; असली कार्यान्वयन प्रक्रिया को कतार में सुला देते हैं और signal पर जगाते हैं — कॉन्टेक्स्ट स्विच की लागत पर।

अभ्यास सेट 5.4

- एक काउंटिंग सेमाफ़ोर 2 से शुरू होता है। पाँच प्रक्रियाएँ बारी-बारी से wait बुलाती हैं। हर कॉल के बाद मान बनाइए और बताइए कौन सी प्रक्रियाएँ रुकी हैं।
- wait का atomic होना क्यों ज़रूरी है — यदि wait खुद आधे में रोका जा सके तो कौन सी दौड़ लौट आती है?

5.5 क्लासिक सिंक्रनाइज़ेशन समस्याएँ

तीन पाठ्यपुस्तक समस्याएँ इस अध्याय की हर कठिनाई को पैकेज करती हैं; OS Lab का Dining Philosophers प्रयोग तीसरी है।

उत्पादक-उपभोक्ता (bounded buffer)। एक उत्पादक प्रक्रिया निश्चित आकार के बफ़र में वस्तुएँ डालती है; उपभोक्ता उन्हें निकालता है। नियम: बफ़र भरने पर उत्पादक प्रतीक्षा करे, बफ़र खाली होने पर उपभोक्ता प्रतीक्षा करे, और दोनों एक ही क्षण बफ़र को कभी न छुएँ। तीन सेमाफ़ोरों से हल: mutex (1, बफ़र की रखवाली), empty (बफ़र आकार से शुरू, खाली स्थान गिनता है), full (0 से शुरू, वस्तुएँ गिनता है)। आपकी प्रिंटर कतार और हर डाउनलोड मैनेजर इसी समस्या के भीतर जीते हैं।

पाठक-लेखक (Readers-Writers)। कई प्रक्रियाएँ एक साझा दस्तावेज़ को साथ मिलकर स्वतंत्र रूप से पढ़ती हैं, लेकिन लेखक को दस्तावेज़ अकेला चाहिए। तनाव: पाठकों को बहने देना लेखक को भूखा रख सकता है; लेखक के लिए ताला लगाना पाठकों की समानांतरता बर्बाद करता है। दो समूहों के बीच न्याय ही पूरी समस्या है।

डाइनिंग फ़िलॉसफ़र्स (Dining Philosophers)। पाँच दार्शनिक गोल मेज़ पर बैठते हैं, हर एक को खाने के लिए दो चॉपस्टिक चाहिए (दोनों ओर एक-एक, पड़ोसियों के साथ साझा)। यदि सब पहले बाई स्टिक उठाएँ, तो सब एक स्टिक पकड़े दूसरे की हमेशा प्रतीक्षा करते हैं — **deadlock** की तस्वीर, जो अध्याय 6 संभालता है। Lab आपसे मेज़ और सुधार लागू करने को कहता है: "दोनों स्टिक साथ उठाओ, वरना प्रतीक्षा करो" और "एक साथ अधिकतम चार बैठने दो" जैसे नियम घरे को तोड़ते हैं।

अभ्यास सेट 5.5

- उत्पादक-उपभोक्ता में, यदि full सेमाफ़ोर भुला दिया जाए तो क्या टूटता है — कौन कब नहीं चलना चाहिए तब चलता है?
- डाइनिंग फ़िलॉसफ़र्स में, कौन सा एक नियम बदलाव हर स्थिति में सब-एक-स्टिक-पकड़े वाली हालत को नष्ट कर देता है?

आम गलतियाँ

- रेस कंडीशन को "कोडिंग बग" कहना। हर प्रक्रिया का कोड ठीक है; दोष साझा डेटा पर बिना सिंक्रनाइज़ किया समय है।
- केवल परस्पर अपवर्जन को शर्त के रूप में गिनना। प्रगति और सीमित प्रतीक्षा के अंक बराबर हैं।
- wait/signal को साधारण बढ़ाना/घटाना लिखना। उनकी atomicता ही पूरा बिंदु है — इसका उल्लेख कीजिए।
- काउंटिंग सेमाफ़ोर को एक संसाधन के ताले से भ्रमित करना; उसका मान गिनता है कि कितने समान संसाधन शेष हैं।

सारांश

- IPC सहयोगी प्रक्रियाओं को डेटा बदलने देता है: साझा मेमोरी (तेज़, स्व-रक्षित) या संदेश भेजना (कर्नेल-माध्यम, सुरक्षित)।
- रेस कंडीशन: साझा डेटा एक साथ बदला गया, परिणाम क्रम पर निर्भर — सही कोड से गलत उत्तर।

- क्रिटिकल सेक्शन साझा डेटा छूने वाला कोड है; सही हल परस्पर अपवर्जन, प्रगति और सीमित प्रतीक्षा देते हैं।
- सेमाफ़ोर atomic wait/signal से ही उपयोग होने वाला OS पूर्णांक है; बाइनरी = एक-एक के लिए mutex, काउंटिंग = समान संसाधनों का समूह।
- उत्पादक-उपभोक्ता (mutex/empty/full), पाठक-लेखक (न्याय), डाइनिंग फ़िलॉसफ़र्स (deadlock झलक) क्लासिक समस्याएँ हैं।

अध्याय अभ्यास सेट

- IPC को परिभाषित कीजिए और उसके दो मॉडल हर एक के एक लाभ के साथ दीजिए।
- एक साझा सीट काउंटर 200 से शुरू होता है; दो बुकिंग प्रक्रियाएँ साथ-साथ एक-एक सीट बेचती हैं और काउंटर 198 के बजाय 199 पर समाप्त होता है। दोष का नाम बताइए और ठीक अंतर्गुथन खोजिए।
- क्रिटिकल-सेक्शन हल की तीन शर्तें बताइए और हर एक को एक-लेन पुल की तस्वीर में समझाइए।
- सेमाफ़ोर क्या है? wait और signal को सटीक परिभाषित कीजिए, और बाइनरी को काउंटिंग से संसाधन-समूह उदाहरण द्वारा अलग कीजिए।
- उत्पादक-उपभोक्ता समस्या और उसके तीन सेमाफ़ोरों की भूमिका का वर्णन कीजिए।
- डाइनिंग फ़िलॉसफ़र्स स्थिति और deadlock रोकने वाला एक नियम समझाइए।
- साझा मेमोरी तेज़ है, तो कोई संदेश भेजना क्यों उपयोग करता है? (दो कारण।)



अध्याय 6: डेडलॉक — शर्तें, रोकथाम, परिहार और बैंकर एल्गोरिदम

(BTEUP CSE तीसरा सेमेस्टर · विषय 3.1 · इकाई 2 — प्रक्रिया प्रबंधन)

आप क्या सीखेंगे

- डेडलॉक क्या है, ठीक-ठीक, यातायात और चॉपस्टिक वाली तस्वीरों के साथ
- वे चार शर्तें जो डेडलॉक के लिए एक साथ मौजूद होनी चाहिए
- चार रणनीतियाँ: अनदेखा करना, रोकथाम, परिहार, पहचान और पुनर्प्राप्ति
- डेडलॉक परिहार और बैंकर एल्गोरिदम, पूरे हल उदाहरण के साथ

6.1 डेडलॉक क्या है?

डेडलॉक वह स्थिति है जिसमें प्रक्रियाओं का एक समूह स्थायी रूप से अवरुद्ध हो जाता है क्योंकि समूह की हर प्रक्रिया एक ऐसा संसाधन धारण किए है जिसकी उसी समूह की दूसरी प्रक्रिया प्रतीक्षा कर रही है, इसलिए कोई भी आगे नहीं बढ़ सकती, कुछ छोड़ नहीं सकती, या समाप्त नहीं हो सकती — अवरोध केवल बाहर से तोड़ा जा सकता है। अध्याय 5 का भोजन यहीं समाप्त हुआ था: हर दार्शनिक एक चॉपस्टिक पकड़े है और दूसरी की प्रतीक्षा कर रहा है जो कभी मुक्त नहीं होगी। एक शहर वही आकृति दिखाता है: कारें चारों ओर से एक चौक को भर देती हैं, हर एक आगे वाली कार की प्रतीक्षा में — जो उसी घेरे में प्रतीक्षा कर रही है।

सेटिंग एक सिस्टम मॉडल है: सिस्टम में संसाधन प्रकार होते हैं (CPU चक्र, मेमोरी स्थान, फाइलें, प्रिंटर, ताले), हर एक की एक या अधिक समान प्रतियाँ। हर प्रक्रिया को हर संसाधन के लिए **माँगना → उपयोग करना → छोड़ना** करना होता है। डेडलॉक उस चक्र का रोग है जब संसाधन न साझा किए जा सकते हैं न वापस लिए जा सकते हैं।

अभ्यास सेट 6.1

- दो प्रक्रियाएँ, एक प्रिंटर: P1 प्रिंटर धारण किए स्कैनर की प्रतीक्षा करती है; P2 स्कैनर धारण किए प्रिंटर की प्रतीक्षा करती है। परिभाषा के अपने शब्दों में बताइए कि यह डेडलॉक क्यों है।
- क्या एक अकेली प्रक्रिया एकल-प्रति संसाधन पर खुद के साथ डेडलॉक कर सकती है? सोचिए, फिर खंड 6.2 से जाँचिए।

6.2 चार आवश्यक शर्तें

डेडलॉक तभी हो सकता है जब **चारों** शर्तें एक ही समय मौजूद हों। किसी एक को हटाइए और डेडलॉक असंभव हो जाता है — यही एक तथ्य पूरे अध्याय को संगठित करता है।

- **परस्पर अपवर्जन (Mutual exclusion)** — एक संसाधन एक समय में केवल एक प्रक्रिया द्वारा धारण किया जा सकता है (प्रिंटर, ताला)।
- **धारण और प्रतीक्षा (Hold and wait)** — कुछ संसाधन धारण किए हुए प्रक्रिया को और की प्रतीक्षा करने की अनुमति है।
- **कोई प्रीएम्पशन नहीं (No preemption)** — संसाधन छीना नहीं जा सकता; केवल धारक ही उसे स्वेच्छा से छोड़ सकता है।
- **वृत्तीय प्रतीक्षा (Circular wait)** — प्रक्रियाओं का एक घेरा मौजूद है, हर एक घेरे में अगले द्वारा धारण किए संसाधन की प्रतीक्षा कर रही है (P1 → P2 → P3 → P1)।

अभ्यास सेट 6.2

- डाइनिंग फ़िलॉसफ़र्स में, दृश्य में चारों शर्तों में से हर एक की ओर इशारा कीजिए।
- एक संसाधन कई प्रक्रियाओं द्वारा एक साथ उपयोग किया जा सकता है (केवल-पढ़ने वाली फाइल)। कौन सी शर्त टूटती है, और डेडलॉक के लिए इसका क्या निष्कर्ष है?

6.3 डेडलॉक सँभालने की चार रणनीतियाँ

1. अनदेखा करना (ostrich दृष्टिकोण)। अधिकांश सामान्य-उद्देश्य सिस्टम, Windows और Linux सहित, आम तौर पर डेडलॉक को सँभालते ही नहीं: यह दुर्लभ है, रोकथाम महँगी है, और अटकाव प्रक्रिया को मारकर या पुनः प्रारंभ करके साफ़ किया जा सकता है। जानबूझकर चुनी गई ईमानदार इंजीनियरिंग।

2. रोकथाम (Prevention) — डिज़ाइन द्वारा एक शर्त तोड़ना। गारंटी दीजिए कि चारों में से कम से कम एक कभी मौजूद नहीं हो सकती:

| तोड़ी गई शर्त | कैसे | लागत | |---|---|---| | परस्पर अपवर्जन | जहाँ संभव हो संसाधन साझा कीजिए (प्रिंटर पर ताला लगाने के बजाय आउटपुट को कतार में स्पूल करना) | सच में न बाँटे जा सकने वाले संसाधनों के लिए असंभव | | धारण और प्रतीक्षा | प्रक्रिया को शुरू होने से पहले अपने सारे संसाधन एक साथ माँगने होंगे | संसाधन धारण किए खाली बैठते हैं; पूरी सूची पहले से ज्ञात होनी चाहिए | | कोई प्रीएम्पशन नहीं | यदि प्रतीक्षारत प्रक्रिया संसाधन धारण किए है तो उन्हें ले लीजिए (अवस्था सहेजकर CPU और मेमोरी के लिए काम करता है) | प्रिंटरों और आधी-लिखी फाइलों के लिए असंभव | | वृत्तीय प्रतीक्षा | सभी संसाधनों को क्रमांक दीजिए; प्रक्रियाएँ केवल बढ़ते क्रम में माँग सकती हैं | क्रम सबके अनुकूल होना चाहिए; कम क्रमांक वाले संसाधनों पर अति-माँग |

3. परिहार (Avoidance) — हर माँग पर सुरक्षित निर्णय। OS हर प्रक्रिया की अधिकतम संभव माँग पहले से जानता है; किसी भी माँग को मंज़ूर करने से पहले वह जाँचता है कि परिणामी अवस्था सुरक्षित है — बैंकर एल्गोरिदम (खंड 6.4)। डिज़ाइन से कुछ भी वर्जित नहीं है; असुरक्षित मंज़ूरियाँ बस मना/विलंबित कर दी जाती हैं।

4. पहचान और पुनर्प्राप्ति (Detection and recovery)। डेडलॉक होने दीजिए, समय-समय पर उसकी जाँच कीजिए (ऐसे घेरे/प्रतीक्षाएँ खोजना जो सुलझ नहीं सकतीं), और घेरे की एक या अधिक प्रक्रियाओं को समाप्त करके या उनके संसाधनों को प्रीएम्प्ट करके पुनर्प्राप्ति कीजिए, सबसे कम काम खोने वाले पीड़ित चुनकर।

अभ्यास सेट 6.3

- एक कॉलेज लैब एकल स्पूल कतार के द्वारा प्रिंट करती है — कौन सी रणनीति और कौन सी शर्त काम कर रही है?
- "सब कुछ पहले से माँगो" किस प्रकार के प्रोग्राम के लिए सबसे स्पष्ट रूप से विफल होता है? एक उदाहरण दीजिए।

6.4 परिहार और बैंकर एल्गोरिदम

सिस्टम अवस्था **सुरक्षित (safe)** है यदि OS फिर भी सभी प्रक्रियाओं को इस प्रकार क्रमबद्ध कर सके कि हर एक अपनी अधिकतम संभव माँग प्राप्त करके समाप्त हो सके — वर्तमान मुक्त संसाधनों के साथ-साथ उन संसाधनों का उपयोग करके जो उससे पहले समाप्त होने वाली प्रक्रियाओं द्वारा छोड़े गए। सुरक्षित होना आज-डेडलॉक-मुक्त दिखने के समान नहीं है: कोई अवस्था आज डेडलॉक के बिना होकर भी **असुरक्षित** हो सकती है, एक दुर्भाग्यपूर्ण माँग की दूरी पर। परिहार सिस्टम को हमेशा सुरक्षित अवस्थाओं के अंदर रखता है।

बैंकर एल्गोरिदम (Dijkstra) वह जाँच करता है। उसका नाम ही तर्क है: बैंकर को नकद इस प्रकार कभी नहीं बाँटना चाहिए कि हर ग्राहक की वादा की गई क्रेडिट सीमा संतुष्ट करना असंभव हो जाए। OS, n प्रक्रियाओं और m संसाधन प्रकारों के लिए, रखता है:

- **Available** — हर संसाधन की वर्तमान में मुक्त प्रतियाँ।
- **Max** — हर प्रक्रिया की अधिकतम कुल माँग (पहले से घोषित)।
- **Allocation** — हर प्रक्रिया वर्तमान में कितना धारण किए है।
- **Need** = $\text{Max} - \text{Allocation}$ — हर प्रक्रिया अभी कितना और माँग सकती है।

सुरक्षा जाँच: $\text{Work} = \text{Available}$; $\text{Finish}[i] = \text{सबके लिए false}$ । बार-बार ऐसी प्रक्रिया खोजिए जिसके लिए $\text{Finish} = \text{false}$ और $\text{Need} \leq \text{Work}$ हो (उसे अभी पूरी तरह संतुष्ट किया जा सकता है); कल्पना कीजिए कि वह समाप्त होती है: $\text{Work} = \text{Work} + \text{उसका Allocation}$, $\text{Finish} = \text{true}$ । यदि हर प्रक्रिया इस प्रकार समाप्त हो सके, तो अवस्था **सुरक्षित** है और मिला क्रम एक सुरक्षित क्रम है। यदि किसी के असमाप्त रहते चक्र रुक जाए, तो अवस्था **असुरक्षित** है।

हल उदाहरण। 5 प्रक्रियाएँ, 3 संसाधन प्रकार (कुल मिलाकर $A = 10$, $B = 5$, $C = 7$ प्रतियाँ)।

प्रक्रिया	Allocation (A B C)	Max (A B C)	Need = Max - Alloc
P0	0 1 0	7 5 3	7 4 3
P1	2 0 0	3 2 2	1 2 2
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1

आवंटित योग: $A = 7$, $B = 2$, $C = 5$, इसलिए **Available** = $(10-7, 5-2, 7-5) = (3, 3, 2)$ ।

सुरक्षा जाँच चलाइए। $\text{Work} = (3, 3, 2)$:

- P1 का $\text{Need} (1, 2, 2) \leq \text{Work} \rightarrow$ P1 समाप्त हो सकती है; $\text{Work} = (3, 3, 2) + (2, 0, 0) = (5, 3, 2)$ ।
- P3 का $\text{Need} (0, 1, 1) \leq \text{Work} \rightarrow$ समाप्त; $\text{Work} = (5, 3, 2) + (2, 1, 1) = (7, 4, 3)$ ।

- P4 का Need $(4, 3, 1) \leq \text{Work} \rightarrow \text{समाप्त}; \text{Work} = (7, 4, 3) + (0, 0, 2) = (7, 4, 5)$ ।
- P0 का Need $(7, 4, 3) \leq \text{Work} \rightarrow \text{समाप्त}; \text{Work} = (7, 4, 5) + (0, 1, 0) = (7, 5, 5)$ ।
- P2 का Need $(6, 0, 0) \leq \text{Work} \rightarrow \text{समाप्त}$ ।

पाँचों समाप्त होते हैं: **अवस्था सुरक्षित है**, एक सुरक्षित क्रम **P1, P3, P4, P0, P2** है। (अन्य सुरक्षित क्रम भी मौजूद हैं; कोई एक सुरक्षा सिद्ध करता है।)

माँग मंजूर करना: जब कोई प्रक्रिया संसाधन माँगती है, तो OS जाँचता है (i) माँग $\leq$ उसकी Need (वरना उसने अपने दावे से अधिक माँगा — त्रुटि), (ii) माँग $\leq$ Available (वरना उसे प्रतीक्षा करनी होगी — संसाधन बस मुक्त नहीं हैं), फिर मंजूरी का नाटक करके काल्पनिक अवस्था पर सुरक्षा जाँच चलाता है: केवल तभी मंजूर कीजिए यदि वह फिर भी सुरक्षित हो। उदाहरण: P1 $(1, 0, 2)$ माँगती है। दोनों जाँचें पास होती हैं (Need $(1, 2, 2)$; Available $(3, 3, 2)$)। नाटक कीजिए: Available $(2, 3, 0)$ हो जाता है, P1 का Allocation $(3, 0, 2)$, Need $(0, 2, 0)$ । Work $(2, 3, 0)$ से सुरक्षा: P1 $(0, 2, 0)$ बैठता है $\rightarrow$ Work $(5, 3, 2)$; फिर पहले की तरह P3, P4, P0, P2 $\rightarrow$ सुरक्षित, इसलिए माँग **मंजूर** होती है। आपका OS Lab ठीक यही लागू करता है: एक प्रयोग में परिहार, उसके साथी में रोकथाम।

असली सिस्टम बैंकर का शायद ही उपयोग क्यों करते हैं: Max माँगें पहले से शायद ही ज्ञात होती हैं, प्रक्रियाएँ और माँगें लगातार बदलती हैं, और जाँच हर माँग पर चलती है। यह परिहार विचार का सबसे स्पष्ट कथन बनकर जीवित है — और उसी कारण बोर्ड का पसंदीदा है।

अभ्यास सेट 6.4

- हल उदाहरण में, क्या P0 किसी सुरक्षित क्रम में सबसे पहले समाप्त हो सकती है? दिखाइए क्यों या क्यों नहीं।
- अब P4 मूल अवस्था से $(3, 3, 0)$ माँगती है। जाँच (i) और (ii) कीजिए; फिर सुरक्षा द्वारा तय कीजिए कि क्या यह मंजूर हो सकती है। (संकेत: सावधानी से नाटक कीजिए — Available $(0, 0, 2)$ हो जाएगा।)

आम गलतियाँ

- चार से कम शर्तें गिनना, या किसी एक को अकेले "कारण" के रूप में पेश करना। चारों एक साथ आवश्यक हैं।
- असुरक्षित अवस्था को "डेडलॉक हुई" कहना। असुरक्षित का अर्थ है डेडलॉक बाद में संभव है; प्रक्रियाएँ अभी चल रही हैं।
- बैंकर में, Need की तुलना कुल संसाधनों से करना, वर्तमान Work/Available के बजाय।

- सुरक्षा चरण का दूसरा आधा भूलना: प्रक्रिया "समाप्त" होने के बाद, उसका Allocation Work में लौटता है।

सारांश

- डेडलॉक: प्रक्रियाओं का समूह जहाँ हर एक वह धारण किए है जिसकी समूह की दूसरी प्रतीक्षा करती है — बाहरी सहायता के बिना स्थायी रूप से अवरुद्ध।
- चार आवश्यक शर्तें, सब एक साथ: परस्पर अपवर्जन, धारण और प्रतीक्षा, कोई प्रीएम्पशन नहीं, वृत्तीय प्रतीक्षा।
- रणनीतियाँ: अनदेखा करना (अधिकांश असली सिस्टम), रोकथाम (डिज़ाइन से शर्त तोड़ना), परिहार (केवल सुरक्षित अवस्थाओं में मंजूरी), पहचान और पुनर्प्राप्ति (घेरा तोड़ने के लिए समाप्त/प्रीएम्प्ट करना)।
- सुरक्षित अवस्था ऐसा क्रम स्वीकार करती है जिसमें हर प्रक्रिया अपने अधिकतम तक पहुँचकर समाप्त हो सके; बैंकर Available, Max, Allocation और $Need = Max - Allocation$ से सुरक्षा जाँचता है।
- हल उदाहरण: 5-प्रक्रिया अवस्था सुरक्षित है (क्रम P1, P3, P4, P0, P2), और P1 की माँग (1, 0, 2) मंजूर-योग्य है।

अध्याय अभ्यास सेट

- डेडलॉक को परिभाषित कीजिए और चार आवश्यक शर्तें, एक-एक लाइन में, बताइए।
- हर शर्त के लिए, एक रोकथाम विधि और उसकी कीमत दीजिए।
- सुरक्षित अवस्था क्या है? असुरक्षित स्थिति के लिए दो-लाइन परिदृश्य द्वारा सुरक्षित, असुरक्षित और डेडलॉक को अलग कीजिए।
- अध्याय के हल उदाहरण का उपयोग करके, जाँचिए कि क्या क्रम P3, P1, P4, P2, P0 भी एक सुरक्षित क्रम है।
- उसी अवस्था से, P0 (0, 2, 0) माँगती है। दोनों जाँचें और सुरक्षा क्रम दिखाते हुए तय कीजिए कि माँग मंजूर हो सकती है या नहीं।
- Windows और Linux व्यवहार में बैंकर एल्गोरिदम क्यों नहीं चलाते? (तीन कारण।)
- पहचान बनाम परिहार: किसे Max पहले से घोषित चाहिए, और कौन खोए हुए काम से कीमत चुकाता है?



अध्याय 7: मेमोरी प्रबंधन और आवंटन — First Fit, Best Fit, Worst Fit

(BTEUP CSE तीसरा सेमेस्टर · विषय 3.1 · इकाई 3 — मेमोरी प्रबंधन)

आप क्या सीखेंगे

- मेमोरी प्रबंधन को क्या हासिल करना चाहिए, और तार्किक बनाम भौतिक एड्रेस का विचार
- सतत आवंटन: निश्चित विभाजन और परिवर्तनशील विभाजन
- स्थान निर्धारण रणनीतियाँ — First Fit, Best Fit, Worst Fit — एक पूरी तुलनात्मक हल के साथ
- Holes (छिद्र), और समय के साथ आवंटन चुनाव क्यों मायने रखता है

7.1 मेमोरी प्रबंधक का काम

RAM तेज़, महँगी और साझा है। मेमोरी प्रबंधक — इस इकाई में अध्ययन किया गया OS भाग — को चाहिए: हर प्रक्रिया को उसकी ज़रूरत की मेमोरी देना, प्रक्रियाओं को एक-दूसरे की मेमोरी छूने से रोकना (सुरक्षा), कौन से भाग उपयोग में हैं और कौन से मुक्त, इसका हिसाब रखना, और हर बाइट का सर्वोत्तम उपयोग करना, क्योंकि जो प्रक्रिया मेमोरी नहीं पा सकती वह चल ही नहीं सकती।

एक विचार इस इकाई की बाकी सब बातें साफ़ कर देता है। प्रोग्राम **तार्किक एड्रेस (logical addresses)** के साथ काम करते हैं — वे एड्रेस जैसे प्रोग्राम उन्हें देखता है, अपनी निजी दुनिया में 0 से शुरू होकर। RAM चिप्स डेटा को **भौतिक एड्रेस (physical addresses)** पर रखती हैं। हार्डवेयर (MMU, मेमोरी प्रबंधन इकाई) हर एक्सेस पर तार्किक → भौतिक अनुवाद करता है, OS द्वारा प्रति प्रक्रिया निर्धारित base/limit मानों का उपयोग करके। इसी प्रकार दो प्रक्रियाएँ दोनों "एड्रेस 1000" का उपयोग

कर सकती हैं और उनका अर्थ अलग असली स्थान होता है — और इसी प्रकार एक प्रक्रिया भौतिक रूप से दूसरी की मेमोरी तक नहीं पहुँच सकती: उसके अनुवाद कभी वहाँ इशारा नहीं करेंगे। इस अध्याय में प्रक्रियाएँ मेमोरी में एकल ब्लॉक के रूप में रखी जाती हैं; अध्याय 8-9 उन्हें पेजों और सेगमेंट में बाँटते हैं।

अभ्यास सेट 7.1

- एक प्रक्रिया का base रजिस्टर 5000 रखता है और वह तार्किक एड्रेस 300 एक्सेस करती है। कौन सा भौतिक एड्रेस छुआ जाता है? यदि वह तार्किक 99999 आज़माए जबकि उसकी limit 8000 है तो क्या होगा?
- एड्रेस अनुवाद मेमोरी सुरक्षा को लगभग मुफ्त क्यों बना देता है?

7.2 सतत आवंटन: दो विभाजन शैलियाँ

सतत आवंटन (contiguous allocation) में हर प्रक्रिया मेमोरी का एक अटूट विस्तार घेरती है। दो क्लासिक शैलियाँ:

- **निश्चित विभाजन (Fixed partitions)** — मेमोरी पहले से तय आकारों के विभाजनों में कटी होती है (मान लीजिए 100 KB, 250 KB, 500 KB, 1 MB); हर प्रक्रिया एक विभाजन लेती है जो उसमें बैठता है। प्रबंधन सरल, लेकिन 250 KB विभाजन में 60 KB प्रक्रिया अपने विभाजन के भीतर 190 KB बर्बाद करती है — **आंतरिक विखंडन (internal fragmentation)**, जिसका अध्याय 9 पूर्ण उपचार करता है।
- **परिवर्तनशील (गतिशील) विभाजन (Variable partitions)** — कोई तय कट नहीं: हर प्रक्रिया को ठीक उतनी जगह दी जाती है जितनी वह माँगती है, मुक्त क्षेत्र से काटकर। सटीक, लेकिन जैसे-जैसे प्रक्रियाएँ लोड होतीं और जाती हैं, मुक्त मेमोरी बिखरे मुक्त ब्लॉक में टूटती है जिन्हें **holes (छिद्र)** कहते हैं, और बर्बाद जगह अब प्रक्रियाओं के बीच पड़ती है — बाहरी विखंडन (वह भी अध्याय 9)। इस अध्याय का शेष भाग इसी शैली में रहता है: जब नई प्रक्रिया आती है और कई छिद्र उसे रख सकते हैं, तो कौन सा छिद्र उसे मिले?

अभ्यास सेट 7.2

- एक 90 KB प्रक्रिया निश्चित 128 KB विभाजन में रखी जाती है। कितनी मेमोरी बर्बाद होती है और उस बर्बादी को क्या कहते हैं?
- परिवर्तनशील विभाजन उसी बर्बादी से क्यों नहीं पीड़ित हो सकते — और उसके स्थान पर कौन सी बर्बादी प्रकट होती है?

7.3 तीन स्थान रणनीतियाँ (हल उदाहरण)

छिद्रों की सूची दी गई हो, तो OS एक रणनीति से एक चुनता है:

- **First Fit** — पहला छिद्र लीजिए (शुरुआत से खोजते हुए, या जहाँ पिछली खोज रुकी थी वहाँ से — वह रूप Next Fit है) जो पर्याप्त बड़ा हो। तेज़: पहली सफलता पर रुकिए।
- **Best Fit** — सभी छिद्र खोजिए और सबसे छोटा वह लीजिए जो बैठता है, सबसे कम शेष छोड़कर।
- **Worst Fit** — सबसे बड़ा छिद्र लीजिए, सबसे बड़ा संभव शेष छोड़कर, इस आशा में कि शेष उपयोगी बना रहेगा।

हल उदाहरण। मुक्त ब्लॉक (छिद्र), मेमोरी क्रम में: **100, 500, 200, 300, 600 KB**। माँगें इस क्रम में आती हैं: **212, 417, 112, 426 KB**। हर रणनीति को एक ही शुरुआती तस्वीर से स्वतंत्र रूप से लगाइए।

First Fit:

- 212 → 500 ब्लॉक (500 पहला है जो बैठता है)। शेष छिद्र 288।
- 417 → 600 ब्लॉक। शेष 183।
- 112 → ऊपर से खोजते हुए: 100 बहुत छोटा; 288 बैठता है → शेष 176।
- 426 → शेष छिद्र हैं 100, 176, 200, 300, 183 — कोई ≥ 426 नहीं। **आवंटित नहीं हो सकता**, कुल 959 KB मुक्त होते हुए भी।

Best Fit:

- 212 → सबसे छोटा बैठने वाला छिद्र 300। शेष 88।
- 417 → सबसे छोटा बैठने वाला 500। शेष 83।
- 112 → सबसे छोटा बैठने वाला 200। शेष 88।
- 426 → सबसे छोटा बैठने वाला 600। शेष 174।
- **चारों आवंटित।**

Worst Fit:

- 212 → सबसे बड़ा छिद्र, 600। शेष 388।
- 417 → अब सबसे बड़ा 500। शेष 83।
- 112 → अब सबसे बड़ा 388। शेष 276।

- 426 → छिद्र हैं 100, 83, 200, 300, 276 — कोई नहीं बैठता। **आवंटित नहीं हो सकता।**

परिणामों में सबक। तीनों रणनीतियों ने एक समान काम का सामना किया। Best Fit ने — बड़ा 600 छिद्र बड़ी 426 माँग के लिए बचाकर — सबको बैठा दिया; First Fit और Worst Fit दोनों ने सबसे बड़ी माँग को कुल मुक्त मेमोरी की प्रचुरता के बीच फँसा दिया। व्यवहार में, असली कार्यभारों के अध्ययन First Fit और Best Fit को व्यापक रूप से तुलनीय पाते हैं (First Fit आमतौर पर तेज़, पूरी खोज की ज़रूरत नहीं; Best Fit के छोटे शेष बेकार झिल्ली बन जाते हैं), और Worst Fit आमतौर पर सबसे कमज़ोर — उसका "बड़ा शेष" वादा इसलिए विफल होता है क्योंकि वह उन बड़े छिद्रों को नष्ट कर देता है जिनकी बड़ी माँगों को ज़रूरत होगी। यह उदाहरण OS Lab का fit-रणनीति प्रयोग भी है, चरण दर चरण।

अभ्यास सेट 7.3

- छिद्र: 150, 350, 80, 420; माँग 300। हर रणनीति कौन सा छिद्र चुनती है, और हर स्थिति में कितना शेष रहता है?
- उसी व्यवस्था में एक अकेली 250 की माँग आती है (300 के Best Fit द्वारा रखा जाने के बाद)। क्या वह रखी जा सकती है? यदि First Fit ने 300 रखा होता तो क्या होता?

7.4 छिद्रों के साथ जीना

आवंटन काम का आधा भाग है; चक्र तब पूरा होता है जब प्रक्रियाएँ जाती हैं। दूसरे छिद्र से सटा मुक्त हुआ ब्लॉक उसमें विलीन हो जाता है (**coalescing / सम्मिलन**) ताकि मुक्त सूची सबसे बड़ी उपयोगी टुकड़ों में रहे। जब कोई एकल छिद्र किसी माँग में न बैठे लेकिन कुल मुक्त मेमोरी बैठती हो, तो OS **संकुचन (compaction)** कर सकता है — सभी प्रक्रियाओं को एक साथ खिसकाकर छिद्रों को एक बड़े ब्लॉक में मिलाना (अध्याय 9) — नकल और ठहराव की लागत पर। ऊपर की रणनीतियाँ सच में एक लंबे खेल के प्रश्न के उत्तर हैं: इस प्रक्रिया को इस प्रकार रखिए कि कल की माँगों के पास फिर भी अवसर रहे।

अभ्यास सेट 7.4

- मेमोरी क्रम: [P1][छिद्र 50][P2][छिद्र 70][P3]। अब P2 जाती है। सम्मिलन से पहले और बाद के मुक्त ब्लॉकों का वर्णन कीजिए।
- संकुचन हर घंटे की दिनचर्या के बजाय अंतिम उपाय क्यों है?

आम गलतियाँ

- तीनों fit को उनके परिणाम के बिना परिभाषाओं के रूप में लिखना: एक ही डेटा पर वे अलग छिद्र आवंटित करते हैं और अलग ढंग से सफल या विफल हो सकते हैं — हल उदाहरण ही अपेक्षित उत्तर का आकार है।
- यह दावा करना कि Best Fit हमेशा सर्वोत्तम है। उसके शेष झिल्लियों में सिकुड़ जाते हैं; लंबे अध्ययन में वह First Fit को समग्र रूप से नहीं हराता।
- यह भूलना कि विफल स्थान (First/Worst Fit में 426) "मेमोरी भरी" नहीं है: कुल मुक्त स्थान माँग से अधिक था — स्थान गलत टुकड़ों में था। यही अवलोकन अध्याय 9 खोलता है।

सारांश

- मेमोरी प्रबंधन: स्थान आवंटित करना, प्रक्रियाओं को एक-दूसरे से बचाना, मुक्त/उपयुक्त क्षेत्रों का हिसाब रखना; तार्किक एड्रेस MMU द्वारा प्रति-प्रक्रिया base/limit का उपयोग करके भौतिक में अनुवादित होते हैं।
- निश्चित विभाजन विभाजनों के अंदर स्थान बर्बाद करते हैं; परिवर्तनशील विभाजन ठीक आकार काटते हैं लेकिन प्रक्रियाओं के बीच छिद्र छोड़ते हैं।
- First Fit पहला पर्याप्त छिद्र लेता है (तेज़); Best Fit सबसे छोटा पर्याप्त (सबसे कम शेष); Worst Fit सबसे बड़ा (सबसे बड़ा शेष, आमतौर पर सबसे कमज़ोर)।
- हल उदाहरण: छिद्र 100/500/200/300/600 पर माँगों 212/417/112/426 के साथ, केवल Best Fit ने चारों को रखा।
- सटे मुक्त छिद्र सम्मिलित होते हैं; संकुचन बिखरे छिद्रों को नकल लागत पर मिला सकता है।

अध्याय अभ्यास सेट

- तार्किक और भौतिक एड्रेस परिभाषित कीजिए। base 12000 और limit 4000 वाली प्रक्रिया तार्किक 2500 और फिर तार्किक 4100 एक्सेस करती है — हर स्थिति में क्या होता है?
- निश्चित और परिवर्तनशील विभाजन की तुलना कीजिए विभाजन आकार, बर्बादी के प्रकार और प्रबंधन प्रयास पर।
- क्रम में छिद्र: 200, 600, 250, 450, 100। माँगें: 380, 120, 290, 500। तीनों रणनीतियों में से हर एक के अंतर्गत स्थान (या विफलता) दिखाइए।

- तीनों fit रणनीतियों को एक-एक लाइन में बताइए, यह नामकरण करते हुए कि हर एक कौन सा छिद्र चुनती है।
- Worst Fit आमतौर पर सबसे कमज़ोर रणनीति क्यों है, उसके समझदारी भरे विचार के बावजूद?
- सम्मिलन (coalescing) क्या है, और वह किस समस्या को टालता है लेकिन हल नहीं करता?





अध्याय 8: स्वैपिंग, पेजिंग और सेगमेंटेशन

(BTEUP CSE तीसरा सेमेस्टर · विषय 3.1 · इकाई 3 — मेमोरी प्रबंधन)

आप क्या सीखेंगे

- स्वैपिंग: पूरी प्रक्रियाओं को RAM और डिस्क के बीच ले जाना
- पेजिंग: मेमोरी को निश्चित आकार के पेजों और फ्रेमों में बाँटना, और पेज टेबल
- पेजिंग में एड्रेस अनुवाद, पूरी तरह हल संख्यात्मक उदाहरण के साथ
- सेगमेंटेशन: प्रोग्राम को तर्क द्वारा बाँटना, और यह पेजिंग से कैसे भिन्न है

8.1 स्वैपिंग

स्वैपिंग किसी पूरी प्रक्रिया को मुख्य मेमोरी से डिस्क के आरक्षित क्षेत्र (स्वैप स्थान) में अस्थायी रूप से हटाना, और निष्पादन जारी रखने के लिए बाद में उसे मेमोरी में वापस लाना है: मध्यकालिक शेड्यूलर (अध्याय 3) किसी प्रक्रिया को तब बाहर swap out करता है जब मेमोरी भीड़ लेती है या प्रक्रिया लंबे समय तक प्रतीक्षा करेगी, और तब अंदर swap in करता है जब मेमोरी खाली होती है और उसकी बारी निकट आती है। स्वैपिंग सभी निवासी प्रक्रियाओं की कुल मेमोरी को भौतिक RAM से अधिक होने देती है — "सिस्टम में" एक साथ जितने कभी समा नहीं सकते उससे अधिक प्रोग्राम रहते हैं। इसकी कीमत समय है: एक प्रक्रिया छवि मेगाबाइट की हो सकती है, और डिस्क स्थानांतरण RAM के मुकाबले धीमे हैं, इसलिए भारी स्वैपिंग अतिभार का लक्षण है (अध्याय 9 इसके सबसे बुरे रूप का नाम देता है)। आधुनिक सिस्टम पेजों के महीन स्तर पर स्वैप करते हैं (खंड 8.2), लेकिन पूरी-प्रक्रिया विचार वह जगह है जहाँ तंत्र — और इकाई 4 का स्वैप स्थान — शुरू होता है।

अभ्यास सेट 8.1

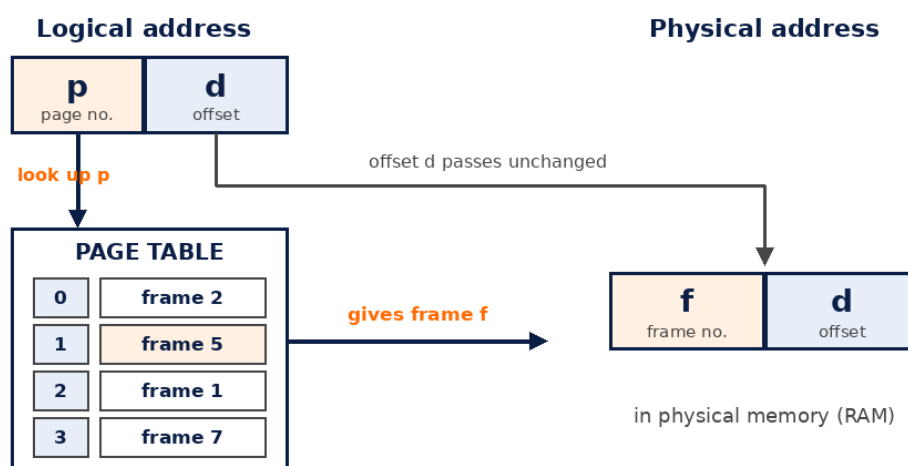
- एक 200 MB प्रक्रिया 100 MB/s चलने वाली डिस्क पर swap out होती है। केवल स्थानांतरण में लगभग कितना समय लगता है, और स्वैप आवृत्ति के बारे में इसका क्या सुझाव है?
- कौन सा शेड्यूलर स्वैपिंग करता है, और कौन सी स्थिति उसे प्रेरित करती है?

8.2 पेजिंग — निश्चित आकार के टुकड़ों में मेमोरी

अध्याय 7 की प्रक्रियाओं को प्रत्येक को एक अटूट ब्लॉक चाहिए था, और छिद्र उसकी कीमत थे। **पेजिंग** वह आवश्यकता हटा देती है: भौतिक मेमोरी निश्चित आकार के ब्लॉक में बँटी होती है जिन्हें **फ्रेम** कहते हैं, और हर प्रक्रिया की तार्किक मेमोरी उसी आकार के ब्लॉक में बँटी होती है जिन्हें **पेज** कहते हैं; कोई पेज किसी भी मुक्त फ्रेम को घेर सकता है, इसलिए प्रक्रिया के पेज RAM में बिखरे पड़े हो सकते हैं जबकि प्रोग्राम फिर भी एक सतत एड्रेस स्पेस देखता है। पेज आकार दो की घात होते हैं (आमतौर पर 4 KB) जो हार्डवेयर चुनता है।

हर तार्किक एड्रेस दो भागों में बँटता है: **पेज संख्या (p)** — प्रक्रिया का कौन सा पेज — और **पेज ऑफ़सेट (d)** — उस पेज के अंदर की स्थिति। OS प्रति प्रक्रिया एक **पेज टेबल** रखता है: पेज संख्या → फ्रेम संख्या (या एक चिह्न जो कहता है कि पेज मेमोरी में नहीं है, जिसे अध्याय 9 पेज फ़ॉल्ट में बदलता है)। प्रति एक्सेस अनुवाद: पेज टेबल में p खोजकर फ्रेम f पाइए; भौतिक एड्रेस f है अपरिवर्तित ऑफ़सेट d के साथ जुड़ा।

Address Translation in Paging



चित्र 8.1 - पेजिंग अनुवाद: पेज संख्या फ्रेम संख्या बनती है; ऑफ़सेट अपरिवर्तित रहता है।

हल उदाहरण। पेज आकार = 100 बाइट (शिक्षण आकार; असली सिस्टम दो की घात उपयोग करते हैं)। एक प्रक्रिया के 16 पेज हैं। उसकी पेज टेबल दिखाती है पेज 7 → फ्रेम 4। तार्किक एड्रेस **753** का अनुवाद कीजिए।

- पेज संख्या $p = 753 \div 100 = 7$; ऑफ़सेट $d = 753 \bmod 100 = 53$ ।
- पेज टेबल: पेज 7 फ्रेम 4 में रहता है।
- भौतिक एड्रेस = फ्रेम $\times$ पेज आकार + ऑफ़सेट = $4 \times 100 + 53 = 453$ ।

दो निष्कर्ष जो परीक्षकों को प्रिय हैं। (1) **कोई बाहरी विखंडन नहीं** — कोई भी मुक्त प्रेम किसी भी पेज की सेवा करता है, इसलिए छिद्र बन ही नहीं सकते; केवल प्रक्रिया का अंतिम पेज आंशिक रूप से खाली हो सकता है, प्रति प्रक्रिया अधिकतम एक पेज की छोटी आंतरिक बर्बादी। (2) पेज टेबल खुद मेमोरी और समय लेती है: 4 KB पेजों के साथ 32-बिट एड्रेस में 12 ऑफ़सेट बिट और 20-बिट पेज संख्या होती है — प्रति प्रक्रिया 2^{20} (लगभग दस लाख) तक प्रविष्टियाँ। असली सिस्टम हाल के अनुवादों के छोटे तेज़ कैश (TLB) और बहु-स्तरीय पेज टेबल से उत्तर देते हैं; अवधारणा अपरिवर्तित रहती है।

अभ्यास सेट 8.2

- पेज आकार 100, पेज टेबल: पेज 2 → फ्रेम 9। तार्किक एड्रेस 260 का अनुवाद कीजिए, चरण दर चरण।
- एक मशीन के पास 100 बाइट के 64 फ्रेम हैं और एक प्रक्रिया 550 बाइट की है। उसे कितने पेज चाहिए, कितनी आंतरिक बर्बादी है, और क्या उसके पेज फ्रेम 3, 7, 9, 20, 21, 40 में बैठ सकते हैं?

8.3 सेगमेंटेशन — तर्क द्वारा बाँटी मेमोरी

कोई प्रोग्राम प्रोग्रामर के मन में एक समान ब्लॉक नहीं होता: वह कोड, ग्लोबल डेटा, स्टैक, हीप, लाइब्रेरी भाग होता है — अलग आकार के टुकड़े जिनके अलग अर्थ और अलग सुरक्षा आवश्यकताएँ हैं। **सेगमेंटेशन** वह मेमोरी योजना है जो उस दृष्टिकोण से मेल खाती है: तार्किक एड्रेस स्पेस **सेगमेंट** में बाँटा होता है, हर एक अपनी लंबाई की एक तार्किक इकाई, जिसे **सेगमेंट संख्या (s)** और उसके अंदर के **ऑफ़सेट (d)** से नामित किया जाता है। OS एक **सेगमेंट टेबल** रखता है: हर सेगमेंट के लिए, उसका **बेस** (प्रारंभिक भौतिक एड्रेस) और उसकी **लिमिट** (लंबाई)। अनुवाद: MMU जाँचता है $d < \text{limit}$ (वरना फ़ॉल्ट — एक्सेस सेगमेंट के अंत से आगे भाग गया) और भौतिक = $\text{base} + d$ गिनता है।

सेगमेंट साझा किए जा सकते हैं (दो प्रक्रियाएँ एक सेगमेंट को उसी कोड की ओर इंगित करती हैं — साझा लाइब्रेरी इसी प्रकार काम करती हैं) और अलग-अलग सुरक्षित किए जा सकते हैं (कोड केवल-पढ़ने योग्य, डेटा पढ़ने-लिखने योग्य), क्योंकि सीमाएँ प्रोग्राम की अपनी हैं। लागत: सेगमेंट भौतिक मेमोरी में परिवर्तनशील आकार के ब्लॉक हैं, इसलिए केवल सेगमेंटेशन अध्याय 7 के छिद्रों और बाहरी विखंडन को वापस लाता है — वह रोग जिसे पेजिंग ने ठीक किया। इसलिए आधुनिक डिज़ाइन दोनों को मिलाते हैं (पेज्ड सेगमेंटेशन: हर सेगमेंट खुद पेजों में बाँटा), भौतिक मेमोरी में बिखराव की समस्या के बिना तार्किक विभाजन रखते हुए।

| आधार | पेजिंग | सेगमेंटेशन | |---|---|---| | विभाजन | निश्चित आकार के पेज, प्रोग्रामर के लिए अदृश्य | परिवर्तनशील आकार की तार्किक इकाइयाँ (कोड, डेटा, स्टैक) | | एड्रेस भाग | पेज संख्या + ऑफ़सेट | सेगमेंट संख्या + ऑफ़सेट | | टेबल प्रविष्टि रखती है | फ्रेम संख्या | बेस + लिमिट | | बाहरी

विखंडन | कोई नहीं | हाँ (परिवर्तनशील ब्लॉक आकार लौटते हैं) | | आंतरिक विखंडन | अंतिम पेज में संभव | कोई नहीं | | साझा करना/सुरक्षा | असुविधाजनक (पेज = मनमाना कट) | स्वाभाविक (सेगमेंट = अर्थपूर्ण इकाई) |

अभ्यास सेट 8.3

- सेगमेंट टेबल: सेगमेंट 1 का बेस 3000, लिमिट 400। ($s=1$, $d=250$), फिर ($s=1$, $d=450$) का अनुवाद कीजिए — दूसरी स्थिति में क्या होता है और यह विशेषता क्यों है?
- कौन सी योजना दो प्रोग्रामों के बीच एक लाइब्रेरी साझा करना स्वाभाविक बनाती है, और क्यों?

आम गलतियाँ

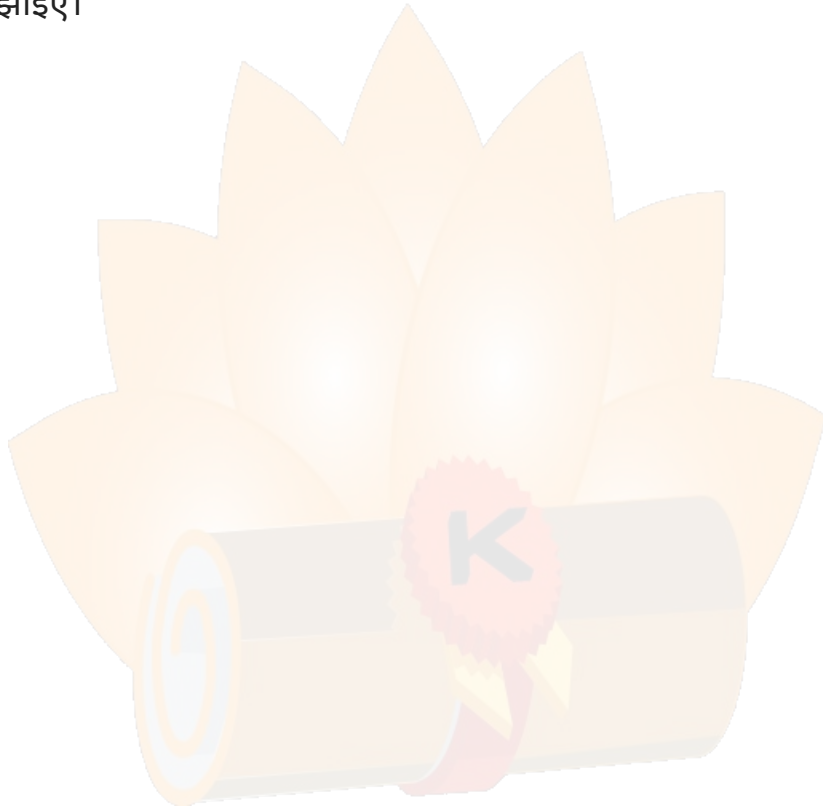
- यह कहना कि पेजिंग में "कोई विखंडन नहीं"। उसमें कोई बाहरी विखंडन नहीं है; अंतिम पेज अंदर स्थान बर्बाद करता है।
- ऑफ़सेट बदला हुआ अनुवाद करना। पेजिंग में ऑफ़सेट कभी नहीं बदलता — केवल पेज संख्या फ्रेम संख्या बनती है।
- सेगमेंटेशन में, लिमिट जाँच भूलना: $d < \text{limit}$ के बिना $\text{base} + d$ गलत भी है और सुरक्षा की हानि भी।
- स्वैपिंग को पेजिंग से भ्रमित करना: स्वैपिंग पूरी प्रक्रियाएँ ले जाती है (या मोटे ढंग से व्यवहार करती है); पेजिंग पेज दर पेज काम करती है और सामान्य आधुनिक तंत्र है।

सारांश

- स्वैपिंग मध्यकालिक शेड्यूलर के अंतर्गत पूरी प्रक्रियाओं को RAM और स्वैप स्थान के बीच ले जाती है, निवासी प्रक्रियाओं को कुल मिलाकर RAM से अधिक होने देती है।
- पेजिंग: निश्चित आकार के पेज किसी भी मुक्त फ्रेम में; पेज टेबल पेज → फ्रेम नक्शा बनाती है; अनुवाद ऑफ़सेट अपरिवर्तित रखता है; हल उदाहरण: तार्किक 753 → भौतिक 453।
- पेजिंग बाहरी विखंडन को मारती है, अंतिम पेज में छोटी आंतरिक बर्बादी रखती है, और पेज टेबल चाहिए (TLB द्वारा सहायता प्राप्त)।
- सेगमेंटेशन तर्क द्वारा बाँटता है (सेगमेंट संख्या + ऑफ़सेट; टेबल लिमिट जाँच के साथ बेस + लिमिट रखती है) — स्वाभाविक साझा करना/सुरक्षा, लेकिन बाहरी विखंडन लौटता है; संयुक्त डिज़ाइन सेगमेंट को पेजों में बाँटते हैं।

अध्याय अभ्यास सेट

- स्वैपिंग परिभाषित कीजिए। स्वैप हुई प्रक्रिया कहाँ जाती है, और उसे कौन वापस लाता है?
- पेज आकार 200 और पेज 5 → फ्रेम 11 के साथ, तार्किक एड्रेस 1130 का पूर्ण अनुवाद कीजिए।
- एक 32-बिट सिस्टम 8 KB पेज उपयोग करता है। ऑफ़सेट कितने बिट का है, और एक प्रक्रिया के पास कितनी पेज-टेबल प्रविष्टियाँ हो सकती हैं?
- पेजिंग को कभी संकुचन (compaction) क्यों नहीं चाहिए?
- सेगमेंट 2: बेस 5000, लिमिट 800 के साथ ($s=2$, $d=100$) का सेगमेंटेशन अनुवाद बनाइए।
- पेजिंग और सेगमेंटेशन की चार बिंदुओं पर तालिका रूप में तुलना कीजिए।
- "पेजिंग प्रोग्राम को बिना पूछे बाँटती है; सेगमेंटेशन प्रोग्राम से पूछती है कि वह कैसे बाँटा है।" तीन लाइनों में समझाइए।





अध्याय 9: पेज फ़ॉल्ट, पेज प्रतिस्थापन, विखंडन और वर्चुअल मेमोरी

(BTEUP CSE तीसरा सेमेस्टर · विषय 3.1 · इकाई 3 — मेमोरी प्रबंधन)

आप क्या सीखेंगे

- पेज फ़ॉल्ट क्या है और उसे सँभालने वाले छह चरण
- तीन प्रतिस्थापन एल्गोरिदम — FIFO, LRU, Optimal — एक हल संदर्भ स्ट्रिंग पर
- Belady की विसंगति: जब अधिक मेमोरी अधिक फ़ॉल्ट देती है
- आंतरिक बनाम बाहरी विखंडन, हमेशा के लिए निपटाया गया
- वर्चुअल मेमोरी और थ्रैशिंग

9.1 पेज फ़ॉल्ट

पेजिंग (अध्याय 8) के साथ, किसी प्रक्रिया के सभी पेजों का RAM में होना ज़रूरी नहीं: पेज टेबल अनुपस्थित पेजों को अमान्य चिह्नित करती है। जब कोई प्रक्रिया अमान्य पेज को छूती है, तो हार्डवेयर **पेज फ़ॉल्ट** उठाता है — कोई त्रुटि नहीं, बल्कि एक अनुरोध: "यह पेज अभी लाओ"। OS उसे छह चरणों में सँभालता है:

- संदर्भ OS के पास ट्रैप होता है; प्रक्रिया निलंबित होती है (वह प्रतीक्षा में जाती है — डिस्क I/O धीमा है)।
- OS जाँचता है कि संदर्भ वैध है (प्रक्रिया के एड्रेस स्पेस के भीतर); यदि नहीं, तो प्रक्रिया समाप्त कर दी जाती है — यह असली त्रुटि स्थिति है।
- एक मुक्त फ्रेम मिलता है — या प्रतिस्थापन के लिए एक पीड़ित पेज चुना जाता है (खंड 9.2) और, यदि पीड़ित बदला गया था, तो पहले डिस्क पर वापस लिखा जाता है।
- आवश्यक पेज डिस्क से फ्रेम में पढ़ा जाता है।
- पेज टेबल अद्यतित होती है: पेज → फ्रेम, वैध चिह्नित।

- फ़ॉल्ट करने वाला निर्देश फिर से शुरू होता है, और प्रक्रिया ऐसे चलती है जैसे कुछ हुआ ही नहीं — सिवाय इसके कि पूरे चक्कर में मिलीसेकंड लगे, सामान्य एक्सेस से हज़ारों गुना अधिक। यही लागत कारण है कि फ़ॉल्टों की संख्या मायने रखती है और अगला खंड मौजूद है।

अभ्यास सेट 9.1

- फ़ॉल्ट करने वाली प्रक्रिया को चलती रखने के बजाय प्रतीक्षा अवस्था में क्यों रखा जाता है?
- यदि चरण 2 संदर्भ को अवैध पाए, तो क्या होता है, और उसे पेज फ़ॉल्ट सफलता की कहानी क्यों नहीं कहते?

9.2 पेज प्रतिस्थापन एल्गोरिदम (हल उदाहरण)

जब कोई फ्रेम मुक्त नहीं होता, किसी को जाना पड़ता है। चुनाव **पेज प्रतिस्थापन एल्गोरिदम** है; इसे **संदर्भ स्ट्रिंग** (प्रक्रिया द्वारा छुए गए पेज नंबरों का क्रम) पर फ़ॉल्ट गिनकर मापा जाता है। एक स्ट्रिंग, तीन एल्गोरिदम, 3 फ्रेम — तीनों हमारे साथ गिनिए।

संदर्भ स्ट्रिंग: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3

FIFO — जो पेज मेमोरी में सबसे लंबे समय से है उसे निकालिए: 7(F), 0(F), 1(F) फ्रेम भरते हैं। 2, 7 को निकालता है (F)। 0 hit। 3, 0 को निकालता है (F)। 0, 1 को निकालता है (F)। 4, 2 को निकालता है (F)। 2, 3 को निकालता है (F)। 3, 0 को निकालता है (F)। 0, 4 को निकालता है (F)। 3, 2, 3 सभी hit। **FIFO फ़ॉल्ट = 10।**

LRU (Least Recently Used) — जो पेज सबसे लंबे समय से अछूता है उसे निकालिए: 7, 0, 1 फ़ॉल्ट से अंदर आते हैं। 2, 7 को निकालता है (सबसे कम ताज़ा) (F)। 0 hit। 3, 1 को निकालता है (F)। 0 hit। 4, 2 को निकालता है (F)। 2, 3 को निकालता है (F)। 3, 0 को निकालता है (F)। 0, 4 को निकालता है (F)। 3, 2, 3 hit। **LRU फ़ॉल्ट = 9।** LRU अतीत को भविष्यवाणी के रूप में पढ़ता है: लंबे समय से अनुपयुक्त पेज संभवतः ठंडा है। इसे हार्डवेयर सहायता चाहिए (काउंटर या संदर्भ स्टैक) और यह क्लासिक व्यावहारिक चुनाव है।

Optimal (OPT) — जिसका अगला उपयोग भविष्य में सबसे दूर है (या कभी नहीं) उसे निकालिए: 7, 0, 1 फ़ॉल्ट से अंदर। 2, 7 को निकालता है (फिर कभी उपयोग नहीं) (F)। 0 hit। 3, 1 को निकालता है (फिर कभी नहीं) (F)। 0 hit। 4, 0 को निकालता है (उसका अगला उपयोग तीनों में सबसे दूर है) (F)। 2, 3 hit। 0, 4 को निकालता है (फिर कभी नहीं) (F)। 3, 2, 3 hit। **Optimal फ़ॉल्ट = 7।** Optimal अजेय है — और लागू नहीं हो सकता, क्योंकि इसे भविष्य चाहिए। इसकी भूमिका मापदंड की है: रिकॉर्ड की गई स्ट्रिंग पर OPT चलाइए और देखिए FIFO/LRU कितने निकट आए।

| एल्गोरिदम | फ़ॉल्ट (3 फ्रेम) | विचार | व्यावहारिक? | |---|---|---|---| | FIFO | 10 | सबसे पुराना पेज बाहर | सरल; Belady विसंगति से पीड़ित | | LRU | 9 | सबसे कम ताज़ा उपयोग बाहर | हाँ, हार्डवेयर सहायता से | | Optimal | 7 | सबसे दूर भविष्य उपयोग बाहर | नहीं — केवल मापदंड |

Belady की विसंगति। अंतर्ज्ञान कहता है अधिक फ्रेम $\Rightarrow$ कम फ़ॉल्ट। FIFO इसे तोड़ सकता है: स्ट्रिंग 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 पर, FIFO 3 फ्रेम के साथ 9 फ़ॉल्ट लेकिन 4 फ्रेम के साथ 10 फ़ॉल्ट देता है (दोनों को अभ्यास सेट में जाँचिए — गिनती ही सीख है)। LRU और Optimal प्रतिरक्षित हैं: वे स्टैक एल्गोरिदम से संबंधित हैं, जिनका n फ्रेम वाला निवासी समूह हमेशा $n+1$ फ्रेम वाले का उपसमूह होता है। यह आपके OS Lab का पेज-प्रतिस्थापन प्रयोग भी है — ठीक ऐसी स्ट्रिंगों पर FIFO, LRU और Optimal।

अभ्यास सेट 9.2

- Belady की विसंगति जाँचिए: 1,2,3,4,1,2,5,1,2,3,4,5 के लिए FIFO फ़ॉल्ट गिनिए 3 फ्रेम के साथ, फिर 4 के साथ।
- अध्याय स्ट्रिंग पर, कौन सा एकल निष्कासन निर्णय LRU को पहले FIFO से बेहतर बनाता है? (दोनों को तब तक ट्रेक कीजिए जब तक वे भिन्न न हों।)
- Optimal असली OS का एल्गोरिदम कभी क्यों नहीं हो सकता, और पाठ्यपुस्तकें उसे क्यों रखती हैं?

9.3 विखंडन — दो बर्बादियाँ, निपटाई गई

विखंडन (Fragmentation) वह मेमोरी बर्बादी है क्योंकि मुक्त स्थान ऐसे टुकड़ों में मौजूद है जिनका आवंटन योजना उपयोग नहीं कर सकती। यह ठीक दो रूपों में आता है, और यह अध्याय अध्याय 7 में खोले गए विषय को बंद करता है:

- **आंतरिक विखंडन (Internal fragmentation)** — आवंटित ब्लॉक के भीतर बर्बादी: ब्लॉक माँग से बड़ा है। निश्चित विभाजन (250 KB विभाजन में 60 KB प्रक्रिया) और पेजिंग का अंतिम पेज (3.2 पेज भरने वाली प्रक्रिया को 4 मिलते हैं) दोनों इससे पीड़ित हैं। स्थान इसलिए खोता है क्योंकि वह किसी का है — मुक्त दिखता है, लेकिन दूसरों के लिए अनुपयोगी।

- **बाहरी विखंडन (External fragmentation)** — आवंटित ब्लॉकों के बीच बर्बादी: कुल मुक्त मेमोरी पर्याप्त है, लेकिन वह बिखरे छिद्रों में पड़ी है, कोई एकल किसी माँग में नहीं बैठता — अध्याय 7 की 426 KB माँग 959 KB मुक्त के बीच मरती हुई। परिवर्तनशील विभाजन और सेगमेंटेशन इससे पीड़ित हैं; पेजिंग नहीं (कोई भी फ्रेम किसी भी पेज में बैठता है)। उपचार: **संकुचन (compaction)** (छिद्रों को मिलाने के लिए प्रक्रियाओं को साथ खिसकाना — महँगी नकल, ठहराव, और I/O मेमोरी को पिन किए हो तो असंभव) या पेजिंग पर जाना, जो उसे पैदा करने वाली आवश्यकता को हटाता है।

अभ्यास सेट 9.3

- निर्णायक विशेषता सहित वर्गीकृत कीजिए: (a) प्रक्रिया के अंतिम पेज के अंत में 18 KB अनुपयुक्त (b) 200 KB प्रक्रिया आने पर पाँच 80 KB छिद्रों के रूप में 400 KB मुक्त।
- एक सिस्टम 30% मेमोरी मुक्त बताता है फिर भी कोई नई प्रक्रिया शुरू नहीं करता। विखंडन की दोनों संभावित व्याख्याएँ दीजिए।

9.4 वर्चुअल मेमोरी — वह अवधारणा जो सब जोड़ती है

वर्चुअल मेमोरी वह मेमोरी योजना है जिसमें हर प्रक्रिया अपने वर्चुअल एड्रेस स्पेस में चलती है जो भौतिक RAM से बड़ा हो सकता है, क्योंकि केवल वर्तमान में आवश्यक पेज मेमोरी में रखे जाते हैं और बाकी डिस्क पर रहते हैं, पेज फ़ॉल्ट पर स्वतः लाए जाते हैं — खंड 9.1–9.2 की तकनीक, सिस्टम डिज़ाइन के रूप में उपयोग की गई, जिसे **डिमांड पेजिंग** कहते हैं। प्रक्रिया के लिए मेमोरी पूरी और विशाल दिखती है; हार्डवेयर के लिए RAM एक कार्यशील उपसमूह रखती है।

इसके लाभ: RAM से बड़े प्रोग्राम चल सकते हैं; एक साथ अधिक प्रक्रियाएँ समा सकती हैं (हर एक खुद का केवल भाग रखती है), मल्टीप्रोग्रामिंग और CPU उपयोग बढ़ाते हुए; शुरुआत में कम I/O चाहिए (माँग पर लोड, सब पहले से नहीं)। इसकी कीमत: पेज फ़ॉल्ट महँगे हैं, इसलिए VM केवल इसलिए काम करती है क्योंकि प्रोग्राम स्थानीयता (locality) दिखाते हैं — किसी छोटे समय खिड़की में वे पेजों का छोटा, स्थिर समूह छूते हैं (लूप, गर्म फंक्शन, निकट डेटा)।

विचार का दुरुपयोग कीजिए और सिस्टम **थ्रैश (thrash)** करता है: जब निवासी प्रक्रियाओं के संयुक्त कार्यशील समूह RAM से अधिक हो जाते हैं, तो हर कुछ निर्देश फ़ॉल्ट करते हैं, डिस्क कतार संतृप्त हो जाती है, CPU उपयोग गिर जाता है (शेड्यूलर खाली प्रक्रियाएँ देखता है और — घातक रूप से — और प्रवेश देता है, इसे बिगाड़ते हुए)। उपचार हैं कार्यशील समूहों को निवासी रखना (प्रति प्रक्रिया पर्याप्त फ्रेम)

या मल्टीप्रोग्रामिंग की मात्रा घटाना: थ्रैशिंग खरीदी गई भौतिक मेमोरी के लिए अत्यधिक वर्चुअल मेमोरी महत्वाकांक्षा है। वक्र याद रखने लायक है: अधिक प्रक्रियाओं के साथ CPU उपयोग बढ़ता है, चरम पर पहुँचता है, फिर थ्रैशिंग पर चट्टान से गिरता है।

अभ्यास सेट 9.4

- 4 GB RAM वाला लैपटॉप ऐसे प्रोग्राम चलाता है जिनका कुल आकार 11 GB है। कौन सी अवधारणा इसे असाधारण नहीं बनाती, और यह प्रोग्रामों के बारे में किस धारणा पर टिकी है?
- CPU उपयोग 8% है जबकि डिस्क 100% पर चलती है। निदान, और दो उपचार।

आम गलतियाँ

- पेज फ़ॉल्ट को त्रुटि या क्रैश कहना। यह वर्चुअल मेमोरी का सामान्य इंजन है; केवल अवैध संदर्भ त्रुटियाँ हैं।
- FIFO में, सबसे कम ताज़ा लोड किए गए को निकालना सही है — लेकिन LRU में उपयोग क्रम के बजाय लोड क्रम से निकालना। Hit LRU के निर्णय को ताज़ा करते हैं; FIFO hit को पूरी तरह अनदेखा करता है।
- विखंडनों को मिलाना: आवंटित ब्लॉक के अंदर (आंतरिक) बनाम मुक्त स्थान में ब्लॉकों के बीच (बाहरी)।
- वर्चुअल मेमोरी को तंत्र के बिना "डिस्क को RAM के रूप में उपयोग करना" परिभाषित करना: फ़ॉल्ट पर डिमांड पेजिंग, प्रति-प्रक्रिया वर्चुअल एड्रेस स्पेस।

सारांश

- पेज फ़ॉल्ट छह चरणों में अनुपस्थित पेज लाता है (ट्रैप → वैधता जाँच → फ्रेम/पीड़ित → पढ़ना → टेबल अद्यतन → निर्देश पुनः प्रारंभ)।
- हल स्ट्रिंग पर 3 फ्रेम के साथ: FIFO 10 फ़ॉल्ट, LRU 9, Optimal 7; केवल FIFO Belady विसंगति से पीड़ित है (3 फ्रेम पर 9 फ़ॉल्ट, 4 पर 10)।
- आंतरिक विखंडन आवंटित ब्लॉकों के अंदर स्थान बर्बाद करता है; बाहरी विखंडन उनके बीच बिखरे छिद्रों में — पेजिंग बाहरी प्रकार को समाप्त करती है।
- वर्चुअल मेमोरी = प्रति-प्रक्रिया वर्चुअल स्पेस पर डिमांड पेजिंग, स्थानीयता पर टिकी; बहुत दूर धकेलने पर यह थ्रैशिंग बन जाती है।

अध्याय अभ्यास सेट

- पेज-फ़ॉल्ट सँभालने के छह चरण क्रम में गिनाइए।
- संदर्भ स्ट्रिंग 1, 2, 3, 2, 1, 4, 3, 3 फ्रेम के साथ: FIFO, LRU और Optimal के अंतर्गत फ़ॉल्ट गिनिए।
- Belady की विसंगति परिभाषित कीजिए और एक एल्गोरिदम परिवार बताइए जो इससे पीड़ित नहीं हो सकता।
- आंतरिक और बाहरी विखंडन को एक-एक उदाहरण के साथ अलग कीजिए, और बताइए कौन सी आवंटन योजनाएँ किससे पीड़ित हैं।
- वर्चुअल मेमोरी क्या है? इसके लाभ और उसके नीचे की स्थानीयता धारणा बताइए।
- CPU-उपयोग वक्र के साथ थ्रैशिंग को शब्दों में समझाइए, और दो उपचार दीजिए।
- बदला गया पीड़ित पेज साफ़ पेज से निकालने में अधिक महँगा क्यों है?





अध्याय 10: मास स्टोरेज संरचना, डिस्क संरचना और डिस्क अटैचमेंट

(BTEUP CSE तीसरा सेमेस्टर · विषय 3.1 · इकाई 4 — I/O सिस्टम)

आप क्या सीखेंगे

- सिस्टम में द्वितीयक (मास) स्टोरेज का स्थान, और उसके मुख्य प्रकार
- हार्ड डिस्क की भौतिक रचना: प्लैटर, ट्रैक, सेक्टर, सिलेंडर
- लॉजिकल ब्लॉक और डिस्क की क्षमता कैसे गिनी जाती है (हल उदाहरण)
- डिस्क एक्सेस समय के तीन भाग (हल उदाहरण)
- डिस्क कैसे जुड़ती हैं: होस्ट-अटैच्ड स्टोरेज, NAS और SAN

10.1 मास स्टोरेज — एक परिचय

मुख्य मेमोरी तेज़ है लेकिन छोटी, महँगी और अस्थिर: RAM में सब कुछ बिजली के साथ मर जाता है। इसलिए सिस्टम को **मास स्टोरेज** (द्वितीयक स्टोरेज) चाहिए — गैर-अस्थिर, बड़ा और प्रति-बाइट सस्ता माध्यम जो OS को खुद, सभी प्रोग्रामों और सभी फाइलों को स्थायी रूप से रखता है, और स्वैप-बाहर किए गए पेजों का घर (अध्याय 8–9) और हर सेव का गंतव्य भी बनता है। मुख्य प्रकार, गति क्रम में:

- **सॉलिड-स्टेट ड्राइव (SSD/NVMe)** — फ्लैश मेमोरी, कोई गतिशील भाग नहीं; सबसे तेज़ द्वितीयक स्टोरेज, अब सिस्टम डिस्क के लिए मानक।
- **हार्ड डिस्क ड्राइव (HDD)** — चुंबकीय घूमते प्लैटर; धीमी लेकिन प्रति टेराबाइट सबसे सस्ती, अब भी बल्क-स्टोरेज की रीढ़, और वह उपकरण जिसकी यांत्रिकी इस इकाई के शेड्यूलिंग विषय (अध्याय 11) को चलाती है।
- **चुंबकीय टेप** — प्रति बाइट बहुत सस्ता, केवल अनुक्रमिक एक्सेस; बैकअप और अभिलेख में जीवित है जहाँ गति का कोई मतलब नहीं।

माध्यम कोई भी हो, OS अधिकतर एक अमूर्तता देखता है: निश्चित आकार के **लॉजिकल ब्लॉक** (आमतौर पर 512 बाइट या 4 KB) की एक सरणी, 0 से क्रमांकित — फाइल सिस्टम (इकाई 5) उस सपाट सरणी पर बना है, और उपकरण अपनी असली ज्यामिति उसके पीछे छिपाता है।

अभ्यास सेट 10.1

- वर्चुअल मेमोरी अकेले RAM में क्यों नहीं रह सकती — वह मास स्टोरेज की कौन सी विशेषता उधार लेती है?
- एक छात्र का लैपटॉप SSD से बूट होता है लेकिन वीडियो HDD पर अभिलेख करता है और बैकअप टेप पर रखता है। हर चुनाव को एक लाइन में न्यायोचित ठहराइए।

10.2 डिस्क संरचना: प्लैटर, ट्रैक, सेक्टर, सिलेंडर

हार्ड डिस्क अपना डेटा चुंबकीय सामग्री से लेपित घूमते गोलाकार **प्लैटर** पर रखती है, हर सतह की सेवा एक गतिशील भुजा पर एक रीड/राइट **हेड** करता है; सभी भुजाएँ साथ चलती हैं। हर सतह संकेंद्रित वलयों में बँटी होती है जिन्हें **ट्रैक** कहते हैं; हर ट्रैक **सेक्टर** में बँटा होता है — स्टोरेज की सबसे छोटी भौतिक इकाई (पारंपरिक रूप से 512 बाइट)। सभी सतहों पर एक ही भुजा स्थिति पर ट्रैक का समूह एक **सिलेंडर** बनाता है। डिस्क पर कोई स्थान इस प्रकार (सिलेंडर, हेड/सतह, सेक्टर) पहचाना जाता है — हालाँकि आधुनिक डिस्क खंड 10.1 की सपाट लॉजिकल-ब्लॉक क्रमांकन पेश करती हैं और आंतरिक रूप से अनुवाद करती हैं।

हल उदाहरण (क्षमता)। एक डिस्क में 4 प्लैटर हैं, दोनों ओर उपयोग = 8 सतहें; प्रति सतह 1024 ट्रैक; प्रति ट्रैक 64 सेक्टर; प्रति सेक्टर 512 बाइट। कुल सेक्टर = $8 \times 1024 \times 64 = 524,288$ । क्षमता = $524,288 \times 512$ बाइट = 268,435,456 बाइट = **256 MB** (बाइनरी इकाइयों में)।

एक्सेस समय। HDD पर डेटा तक पहुँचने में तीन घटक लगते हैं:

- **सीक समय (Seek time)** — भुजा को सही ट्रैक पर ले जाना (सबसे धीमा, सबसे परिवर्तनशील भाग)।
- **घूर्णन विलंब (Rotational latency)** — प्लैटर के घूमकर वांछित सेक्टर को हेड के नीचे लाने की प्रतीक्षा; औसतन, आधा घूर्णन। 6000 RPM पर एक घूर्णन $60/6000$ सेकंड = 10 ms लेता है, इसलिए औसत विलंब = **5 ms**।
- **स्थानांतरण समय (Transfer time)** — सेक्टरों के हेड के नीचे से गुजरने पर सच में पढ़ना/लिखना; छोटे स्थानांतरणों के लिए छोटा।

डिस्क शेड्यूलिंग (अध्याय 11) पहले घटक पर हमला करती है: जिस क्रम में माँगें पूरी होती हैं वह तय करता है कि भुजा कितनी दूर यात्रा करती है।

अभ्यास सेट 10.2

- एक डिस्क: 6 सतहें, 2048 ट्रैक/सतह, 128 सेक्टर/ट्रैक, 512 B/सेक्टर। इसकी क्षमता GB (बाइनरी) में गिनीए।
- एक 15000 RPM डिस्क: औसत घूर्णन विलंब क्या है? "औसत" क्यों, ठीक-ठाक क्यों नहीं?

10.3 डिस्क अटैचमेंट — स्टोरेज कैसे जुड़ता है

- **होस्ट-अटैच स्टोरेज** — डिस्क उसके I/O बसों के द्वारा कंप्यूटर से सीधे जुड़ी (ऐतिहासिक रूप से IDE/PATA, आज HDD/SSD के लिए SATA और तेज़ SSD के लिए PCIe पर NVMe), एक होस्ट **कंट्रोलर** द्वारा प्रबंधित जिसे OS अपने डिवाइस ड्राइवर के द्वारा चलाता है। आपके लैब PC की आंतरिक डिस्क यही हैं। छोटा पैमाना, सरल, स्टोरेज उसी एक मशीन की होती है।
- **नेटवर्क-अटैच स्टोरेज (NAS)** — नेटवर्क पर रखा एक स्टोरेज उपकरण जो कई क्लाइंट को नेटवर्क पर फाइलें देता है (साझा कार्यालय ड्राइव); क्लाइंट फोल्डर देखते हैं, कच्चे ब्लॉक नहीं।
- **स्टोरेज-एरिया नेटवर्क (SAN)** — ब्लॉक स्टोरेज का एक समर्पित उच्च-गति नेटवर्क जिससे कई सर्वर इस प्रकार जुड़ते हैं जैसे डिस्क स्थानीय हों; सर्वर कच्चे लॉजिकल ब्लॉक देखते हैं और अपने फाइल सिस्टम खुद बनाते हैं। संस्थागत पैमाने पर डेटा केंद्रों में उपयोग होता है।

परीक्षा का अंतर: NAS साधारण नेटवर्क पर फाइलें निर्यात करता है; SAN समर्पित स्टोरेज नेटवर्क पर ब्लॉक निर्यात करता है; होस्ट-अटैच कुछ निर्यात नहीं करता — वह एक होस्ट से तारबद्ध होता है।

अभ्यास सेट 10.3

- एक डेटा केंद्र 40 सर्वरों को डिस्क का एक साझा समूह देता है जिसे हर सर्वर खुद फॉर्मेट करता है। NAS या SAN — और किस सुराग ने फैसला किया?
- OS को SATA डिस्क और NAS बॉक्स के लिए अलग ड्राइवर कहानी क्यों चाहिए?

10.4 फॉर्मेटिंग: कच्ची डिस्क से उपयोगी स्टोरेज तक

कच्ची डिस्क तीन चरणों में उपयोगी बनती है: **भौतिक (निम्न-स्तरीय) फॉर्मेटिंग** प्लैटर को उनके हेडर और त्रुटि-सुधार कोड के साथ सेक्टरों में बाँटती है (निर्माता द्वारा की गई); **विभाजन (partitioning)** डिस्क को क्षेत्रों में काटता है जिन्हें OS अलग डिस्क मानता है (C:, D:...); **लॉजिकल फॉर्मेटिंग** एक विभाजन में फाइल सिस्टम लिखती है (इकाई 5 का विषय) ताकि फाइलें मौजूद हो सकें। तभी सपाट लॉजिकल-ब्लॉक सरणी रोज़मर्रा की फाइलें और फोल्डर बनती है।

अभ्यास सेट 10.4

- एक बिल्कुल नई डिस्क OS इंस्टॉलर में "unallocated" दिखती है। कौन से फॉर्मेटिंग चरण शेष हैं, और किस क्रम में?

आम गलतियाँ

- सेक्टर को लॉजिकल ब्लॉक परिभाषित करना। सेक्टर भौतिक इकाई है; OS का लॉजिकल ब्लॉक सेक्टरों को फैला या उनसे मेल खा सकता है — उपकरण ब्लॉक पेश करता है, प्लैटर सेक्टर रखता है।
- यह भूलना कि घूर्णन विलंब केवल RPM से मापा जाता है: सीक समय भुजा की गति के बारे में है, घूर्णन गति के बारे में नहीं।
- यह कहना कि NAS और SAN दोनों "नेटवर्क पर स्टोरेज" हैं, फाइल-बनाम-ब्लॉक अंतर के बिना — वही अंतर उत्तर है।

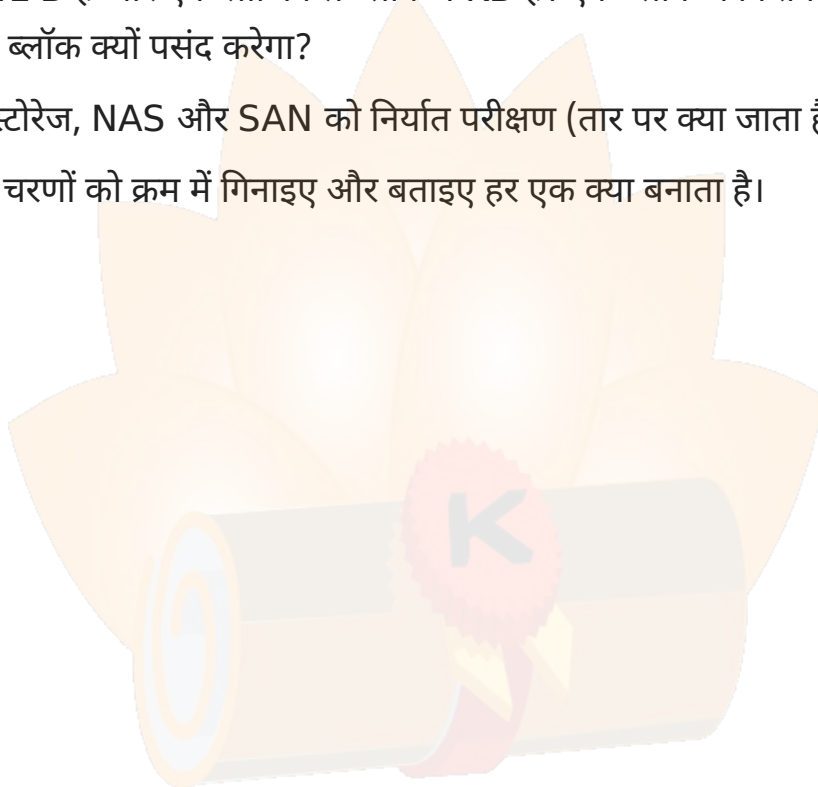
सारांश

- मास स्टोरेज गैर-अस्थिर, बड़ा और प्रति बाइट सस्ता है: SSD (सबसे तेज़), HDD (बल्क), टेप (बैकअप); OS लॉजिकल ब्लॉक की सरणी देखता है।
- HDD रचना: प्लैटर/सतहें, एक भुजा पर हेड, ट्रैक, सेक्टर, सिलेंडर। क्षमता = सतहें × ट्रैक × सेक्टर × बाइट (हल: 256 MB)।
- एक्सेस समय = सीक + घूर्णन विलंब (औसत = आधा घूर्णन; 6000 RPM पर 5 ms) + स्थानांतरण।
- अटैचमेंट: होस्ट-अटैच्ड (एक मशीन को SATA/NVMe), NAS (LAN पर फाइलें), SAN (समर्पित नेटवर्क पर ब्लॉक)।

- उपयोगी स्टोरेज को भौतिक फॉर्मेटिंग, विभाजन, फिर फाइल सिस्टम के साथ लॉजिकल फॉर्मेटिंग चाहिए।

अध्याय अभ्यास सेट

- तीन मुख्य मास-स्टोरेज प्रकारों के नाम बताइए, हर एक की एक शक्ति, और RAM के विरुद्ध सबकी साझा विशेषता।
- 10 सतहों, प्रति सतह 4096 ट्रैक, प्रति ट्रैक 256 सेक्टर, प्रति सेक्टर 512 बाइट वाली डिस्क की क्षमता गिनिए (उत्तर GB, बाइनरी में)।
- सीक समय, घूर्णन विलंब और स्थानांतरण समय परिभाषित कीजिए। डिस्क शेड्यूलिंग किसे घटाती है, और कैसे (एक लाइन, अध्याय 11 की ओर इशारा करते हुए)?
- एक सेक्टर 512 B है और एक लॉजिकल ब्लॉक 4 KB है। एक ब्लॉक में कितने सेक्टर, और OS सेक्टरों से बड़े ब्लॉक क्यों पसंद करेगा?
- होस्ट-अटैच्ड स्टोरेज, NAS और SAN को निर्यात परीक्षण (तार पर क्या जाता है) से अलग कीजिए।
- तीन फॉर्मेटिंग चरणों को क्रम में गिनाइए और बताइए हर एक क्या बनाता है।





अध्याय 11: डिस्क शेड्यूलिंग, स्वैप स्थान प्रबंधन और RAID

(BTEUP CSE तीसरा सेमेस्टर · विषय 3.1 · इकाई 4 — I/O सिस्टम)

आप क्या सीखेंगे

- डिस्क माँगों का क्रम प्रदर्शन क्यों तय करता है
- एक ही हल कतार पर FCFS, SSTF, SCAN और C-SCAN, कुल हेड गति की गणना सहित
- स्वैप स्थान: यह कहाँ रहता है और OS इसे कैसे प्रबंधित करता है
- RAID: कई डिस्क एक के रूप में — स्तर 0, 1, 5, 6 और 10

11.1 डिस्क शेड्यूलिंग — समस्या और माप

डिस्क ब्लॉक की माँगें एक साथ कई प्रक्रियाओं से आती हैं और कतार में लगती हैं। हार्ड डिस्क पर प्रमुख लागत **सीक** है: भुजा को हिलाना (अध्याय 10)। इसलिए जिस क्रम में कतारबद्ध माँगें पूरी होती हैं वह कुल भुजा यात्रा तय करता है — और उसके साथ, डिस्क की प्रभावी गति। **डिस्क शेड्यूलिंग** OS द्वारा प्रतीक्षारत माँग कतार को पुनः क्रमित करना है; इसे दी गई कतार पर **कुल हेड गति** (पार किए गए सिलेंडरों में) से मापा जाता है, कम बेहतर। निष्पक्षता भी मायने रखती है: यात्रा न्यूनतम करने वाला एल्गोरिदम भी स्वीकार्य नहीं यदि वह दूर की माँग को अनिश्चित काल तक भूखा रख सके। SSD में कोई भुजा नहीं होती, इसलिए यह विषय HDD का विषय है — अध्याय 10 में HDD यांत्रिकी भरने का एक और कारण।

पूरे खंड की हल कतार: सिलेंडरों **98, 183, 37, 122, 14, 124, 65, 67** की माँगें, उसी आगमन क्रम में; हेड सिलेंडर **53** से शुरू होता है; डिस्क में सिलेंडर **0-199** हैं। (यह OS Lab का डिस्क-शेड्यूलिंग प्रयोग है, कागज़ पर।)

FCFS — आगमन क्रम में सेवा

सेवा क्रम: 53 → 98 → 183 → 37 → 122 → 14 → 124 → 65 → 67। गतियाँ: 45 + 85 + 146 + 85 + 108 + 110 + 59 + 2 = **640 सिलेंडर**। निष्पक्ष — कोई कभी कतार नहीं तोड़ता — लेकिन भुजा डिस्क भर में ज़िगज़ैग करती है; यह हराने योग्य आधाररेखा है।

SSTF — सबसे कम सीक समय पहले

हमेशा वर्तमान हेड स्थिति के सबसे निकट प्रतीक्षारत माँग की सेवा कीजिए। 53 से: 65 (12), 67 (2), फिर 37 (30), 14 (23), फिर पार करके 98 (84), 122 (24), 124 (2), 183 (59)। कुल = **236 सिलेंडर**। कहीं बेहतर यात्रा — लेकिन मध्य माँगों की स्थिर धारा किनारों की माँगों को भूखा रख सकती है, ठीक अध्याय 4 के SJF का रोग डिस्क के वस्त्रों में।

SCAN — लिफ्ट (elevator)

भुजा एक दिशा में झाड़ू लगाती है, रास्ते की हर चीज़ की सेवा करती हुई, डिस्क के अंत तक जाती है, उलटती है, और वापस झाड़ू लगाती है — मंज़िलों की सेवा करने वाली लिफ्ट की तरह। मानिए हेड बड़े सिलेंडर नंबरों की ओर बढ़ रहा है: क्रम: 65, 67, 98, 122, 124, 183, (अंत 199), फिर वापस: 37, 14। गतियाँ: 12 + 2 + 31 + 24 + 2 + 59 + 16 (199 तक) + 162 (37 तक नीचे) + 23 = **331 सिलेंडर**। कोई भुखमरी नहीं: हर माँग एक पूरे स्वीप के भीतर पहुँची जाती है। इसका सापेक्ष **LOOK** भौतिक अंत तक जाने के बजाय अंतिम माँग पर मुड़ जाता है।

C-SCAN — वृत्तीय SCAN

केवल एक दिशा में सेवा कीजिए; अंत में, शुरुआत पर कूदिए और फिर झाड़ू लगाइए (वापसी की कूद यहाँ गति गिनी जाती है; कुछ पाठ उसे छोड़ देते हैं — अपना नियम बताइए)। क्रम: 65, 67, 98, 122, 124, 183, (199), 0 पर कूद, फिर 14, 37। गतियाँ: 146 (199 तक ऊपर) + 199 (कूद) + 14 + 23 = **382 सिलेंडर**। C-SCAN का लाभ कुल यात्रा नहीं बल्कि समानता है: माँगों को एक वृत्त माना जाता है, इसलिए प्रतीक्षा समय SCAN की तुलना में अधिक समान रूप से फैलता है (भुजा के ठीक पीछे की माँगें पूरा दोहरा स्वीप नहीं झेलतीं)। **C-LOOK** वही विचार है जिसमें कूद एक ओर की अंतिम माँग से दूसरी ओर की पहली माँग तक ली जाती है।

| एल्गोरिदम | कुल हेड गति | भुखमरी? | विचार | |---|---|---|---| | FCFS | 640 | नहीं |
आगमन क्रम | | SSTF | 236 | हाँ (किनारे) | सबसे निकट माँग पहले | | SCAN | 331 | नहीं |
दोनों ओर लिफ्ट स्वीप | | C-SCAN | 382 | नहीं | एक-दिशा स्वीप, वृत्तीय |

अभ्यास सेट 11.1

- कतार 82, 170, 43, 140, 24, 16, 190, हेड 50 पर (डिस्क 0–199): FCFS कुल गति गिनिए, फिर SSTF का सेवा क्रम।
- SSTF की भुखमरी असली OS को उसकी यात्रा बचत से अधिक क्यों चिंतित करती है?

11.2 स्वैप स्थान प्रबंधन

अध्याय 8–9 पेजों और प्रक्रियाओं को डिस्क पर भेजते रहे: **स्वैप स्थान** वह जगह है जहाँ वे रहते हैं — मेमोरी के विस्तार के रूप में शुद्ध रूप से आरक्षित डिस्क क्षेत्र। यह OS द्वारा विशेष रूप से प्रबंधित होता है, साधारण फाइल सिस्टम के द्वारा नहीं, दो कारणों से: गति (स्वैप यातायात को डायरेक्ट्री और आवंटन जैसे फाइल-सिस्टम अतिरिक्त भार को बायपास करना चाहिए; OS इसके बजाय स्वैप स्लॉट का एक सरल नक्शा रखता है) और उद्देश्य (इसकी सामग्री परिभाषा से अस्थायी है — स्वैप में किसी चीज़ को नाम, साझा करना या बैकअप नहीं चाहिए)। दो स्थान मौजूद हैं: **समर्पित स्वैप विभाजन** (एक कच्चा डिस्क क्षेत्र, क्लासिक Linux तरीका — सबसे तेज़, निश्चित आकार) और सामान्य फाइल सिस्टम के अंदर एक **स्वैप फाइल** (बनाने/आकार बदलने में लचीली, थोड़ी धीमी)। कितना आरक्षित करें, यह एक निर्णय है: बहुत कम तो भारी भार पूरी तरह मेमोरी से बाहर हो जाता है; बहुत अधिक डिस्क बर्बाद करता है — आधुनिक मार्गदर्शन RAM आकार और कार्यभार के साथ मापता है, और हाइबरनेशन उपयोग करने वाले सिस्टम को कम से कम RAM जितना बड़ा स्वैप चाहिए क्योंकि पूरी मेमोरी छवि वहाँ लिखी जाती है।

अभ्यास सेट 11.2

- OS स्वैप हुए पेजों को साधारण फाइलों के रूप में क्यों नहीं रखता?
- एक सिस्टम कभी हाइबरनेट नहीं करता लेकिन हर रात थ्रैश करता है। क्या उसकी समस्या स्वैप आकार है? दो लाइनों में बचाव कीजिए (अध्याय 9 याद कीजिए)।

11.3 RAID — कई डिस्क एक के रूप में कार्य करती हुई

RAID (Redundant Array of Independent/Inexpensive Disks) कई भौतिक डिस्क को एक तार्किक इकाई में मिलाता है ताकि वह मिले जो एकल डिस्क नहीं दे सकती: उच्च **प्रदर्शन** (डेटा को डिस्क में बाँटना — striping — ताकि भाग समानांतर में पढ़े जाएँ), उच्च **विश्वसनीयता** (अतिरिक्त जानकारी रखना ताकि डिस्क की विफलता कुछ न खोए), या दोनों। मानक स्तर:

स्तर	तकनीक	डिस्क (न्यूनतम)	डिस्क विफलता से बचता है?	स्थान कुशलता	इसका बिंदु
RAID 0	केवल striping	2	**नहीं** — विफलता सब कुछ खोती है	100%	शुद्ध गति; स्ट्रेच/वीडियो कार्य
RAID 1	मिररिंग (हर डिस्क की प्रतिलिपि)	2	हाँ (एक दर्पण शेष)	50%	सरल विश्वसनीयता; OS डिस्क
RAID 5	Striping + एक वितरित पैरिटी	3	हाँ (कोई एक डिस्क)	$(n-1)/n$	संतुलित: सर्वरों का क्लासिक
RAID 6	Striping + दो पैरिटी	4	हाँ (कोई दो डिस्क)	$(n-2)/n$	सुरक्षा, जैसे डिस्क विशाल होती हैं
RAID 10	दर्पण युग्म, striped (1+0)	4	हाँ (प्रति युग्म एक)	50%	गति और सुरक्षा; डेटाबेस

पैरिटी 5 और 6 के पीछे की अंकगणितीय चतुराई है: डिस्क में डेटा ब्लॉक की हर पट्टी के लिए, एक अतिरिक्त ब्लॉक उनका XOR (एक चेकसम) रखता है; यदि कोई एक डिस्क मर जाए, तो उसके ब्लॉक बचे हुआ और पैरिटी से फिर गिने जाते हैं — सरणी काम करती रहती है, घटी हुई, और प्रतिस्थापन डिस्क पर पुनर्निर्माण करती है। RAID 2, 3 और 4 ऐतिहासिक रूप हैं (बिट-स्तरीय striping, समर्पित पैरिटी डिस्क) जिनके नाम पाठ्यक्रम की "RAID के प्रकार" आपसे अपेक्षा करती है, जबकि 0, 1, 5, 6, 10 असली उपयोग के स्तर हैं। एक चेतावनी जो हर प्रशासक दोहराता है: **RAID बैकअप नहीं है** — यह डिस्क विफलता से बचाता है, विलोपन, भ्रष्टाचार, चोरी या आग से नहीं, जो सभी प्रतियाँ एक साथ ले जाते हैं।

अभ्यास सेट 11.3

- 2 TB डिस्क की 4-डिस्क RAID 5 सरणी: कितना उपयोगी स्थान, और एक साथ कितनी डिस्क विफल हो सकती हैं?
- वही डिस्क RAID 10 में: उपयोगी स्थान, विफलता सहनशीलता, और RAID 5 पर उसे फिर भी चुनने का एक कारण।

आम गलतियाँ

- SCAN/C-SCAN में केवल सेवा की गई माँगों के बीच हेड गति गिनना और अंत तक की यात्रा (और C-SCAN की वापसी) भूलना — या उलटा, जब प्रश्न का नियम LOOK हो। पढ़िए कौन सा रूप पूछा गया है।
- SSTF को "सर्वोत्तम" कहना — यह लालची है और भूखा रखता है; SCAN परिवार गारंटी के लिए थोड़ी यात्रा का व्यापार करता है।
- RAID 0 को विश्वसनीय परिभाषित करना क्योंकि वह RAID है। स्तर 0 में शून्य अतिरेक है — RAID का R अपने सबसे प्रसिद्ध स्तर पर विफल होता है।
- "RAID एक बैकअप सिस्टम है" लिखना। यह केवल एक विफलता प्रकार से बचाता है।

सारांश

- डिस्क शेड्यूलिंग HDD पर सीक यात्रा घटाने के लिए माँग कतार को पुनः क्रमित करती है; कुल हेड गति और निष्पक्षता से मापा जाता है।
- हल कतार (हेड 53): FCFS 640, SSTF 236 (भूखा रखता है), SCAN 331, C-SCAN 382 — स्वीप भुखमरी-रहित और समान प्रतीक्षा के लिए यात्रा का व्यापार करते हैं।
- स्वैप स्थान मेमोरी अतिप्रवाह के लिए आरक्षित OS-प्रबंधित डिस्क है — विभाजन या फाइल, गति के लिए फाइल सिस्टम को बायपास करती हुई।
- RAID सरणियाँ गति के लिए stripe करती हैं और जीवित रहने के लिए दर्पण/पैरिटी: 0 केवल गति, 1 दर्पण, 5 एक पैरिटी, 6 दो पैरिटी, 10 दर्पण पट्टियाँ; RAID बैकअप नहीं है।

अध्याय अभ्यास सेट

- कतार: 55, 120, 18, 88, 160, 42, हेड 40 पर (डिस्क 0-199)। FCFS, SSTF और SCAN (हेड ऊपर बढ़ता हुआ, अंत तक) के लिए कुल हेड गति गिनिए।
- उसी कतार के लिए, कुल के बिना C-LOOK का सेवा क्रम दीजिए (हेड ऊपर बढ़ता हुआ)।
- SSTF भुखमरी को उसे प्रदर्शित करने वाले माँग प्रतिरूप से समझाइए।
- स्वैप स्थान क्या है? स्वैप विभाजन और स्वैप फाइल की गति और लचीलेपन पर तुलना कीजिए।
- 5-डिस्क सरणी, हर एक 4 TB: RAID 5 और RAID 6 के अंतर्गत उपयोगी स्थान और एक साथ विफलता सहनशीलता।

- पैरिटी मृत डिस्क का डेटा पुनर्निर्माण क्यों कर सकती है? XOR विचार से दो लाइनों में उत्तर दीजिए।
- "हमारे सर्वर में RAID 1 है, इसलिए हमें बैकअप नहीं चाहिए।" कथन को पूरी तरह सुधारिए।





अध्याय 12: फाइल अवधारणा, एक्सेस विधियाँ, डायरेक्टरी संरचना और माउंटिंग

(BTEUP CSE तीसरा सेमेस्टर · विषय 3.1 · इकाई 5 — फाइल प्रबंधन)

आप क्या सीखेंगे

- OS की दृष्टि से फाइल क्या है, और उसके गुण व कार्य
- दो एक्सेस विधियाँ: अनुक्रमिक और प्रत्यक्ष
- डायरेक्टरी संरचनाएँ, एकल-स्तर से उस वृक्ष तक जो आप रोज़ उपयोग करते हैं
- निरपेक्ष और सापेक्ष पथ, और फाइल सिस्टम को माउंट करने का अर्थ

12.1 फाइल अवधारणा और उसके गुण

डिस्क कच्चे ब्लॉक पेश करती हैं (अध्याय 10); मनुष्य दस्तावेज़ों, फोटो और प्रोग्रामों में सोचते हैं। सेतु **फाइल** है: संबंधित जानकारी का एक नामित संग्रह जो द्वितीयक स्टोरेज पर एक तार्किक इकाई के रूप में रखा जाता है, जिसके साथ उपयोगकर्ता और प्रोग्राम ब्लॉक और सेक्टर के बारे में कुछ जाने बिना काम करते हैं। फाइल अपने निर्माता की सामग्री से परिभाषित होती है — पाठ, डेटा, स्रोत कोड, निष्पादन योग्य, छवि — और OS उसे निश्चित लेखा-जोखा, **फाइल गुणों**, के द्वारा समर्थन देता है:

- **नाम** — मनुष्य-उन्मुख पहचानकर्ता (वह एकमात्र गुण जो लोग आमतौर पर देखते हैं)।
- **पहचानकर्ता (Identifier)** — एक अद्वितीय आंतरिक टैग (एक संख्या) जिसके द्वारा OS फाइल को जानता है।
- **प्रकार** — पाठ, निष्पादन योग्य, डायरेक्टरी...; एक्सटेंशन (.txt, .exe, .jpg) से संकेतित या फाइल में संग्रहीत।
- **स्थान** — उपकरण पर उसका डेटा कहाँ रहता है (वह सूचक जिसे अध्याय 13 विस्तार देता है)।
- **आकार** — बाइट में वर्तमान लंबाई (और कभी-कभी अधिकतम अनुमत)।
- **सुरक्षा** — कौन पढ़, लिख, निष्पादित कर सकता है (अध्याय 13)।

- **समय, तारीख और स्वामी** — बनाई/बदली/एक्सेस की गई मुहरें और वह उपयोगकर्ता जो उसका स्वामी है।

वे **कार्य** जो हर OS फाइलों पर देता है: create (बनाना), write (लिखना), read (पढ़ना), reposition (फाइल सूचक को हिलाना — फाइल कार्य का seek), delete (मिटाना), और truncate (फाइल रखना, उसकी सामग्री खाली करना); open और close इन्हें लपेटते हैं — खोलना फाइल के गुणों को एक खुली-फाइल तालिका में लोड करता है ताकि बार-बार के कार्य तेज़ रहें, और प्रति-प्रक्रिया सूचक ट्रैक करता है कि हर खुले सत्र ने कितना पढ़ा या लिखा है।

अभ्यास सेट 12.1

- OS सच में फाइल खोजने के लिए किस गुण का उपयोग करता है, यदि नाम नहीं? एक लाइन में नाम → पहचानकर्ता → स्थान खोजिए।
- एक ही फाइल दो प्रोग्रामों में खोलना: कितनी खुली-फाइल प्रविष्टियाँ, और उनके सूचक कैसे व्यवहार करते हैं?

12.2 एक्सेस विधियाँ

खुलने के बाद, फाइल के बाइट दो क्लासिक प्रतिरूपों में उपभोग किए जा सकते हैं:

- **अनुक्रमिक एक्सेस (Sequential access)** — जानकारी क्रम में संसाधित होती है, एक रिकॉर्ड/बाइट दूसरे के बाद, शुरुआत से (या जहाँ सूचक खड़ा है वहाँ से), जैसे किताब पेज दर पेज पढ़ना या टेप चलाना। पढ़ना और लिखना सूचक को स्वतः आगे बढ़ाते हैं। यह लॉग, मीडिया प्लेबैक, स्रोत फाइल संकलन और अधिकांश रोज़मर्रा कार्य का स्वाभाविक प्रतिरूप है — और टेप द्वारा अनुमत एकमात्र प्रतिरूप।
- **प्रत्यक्ष (यादृच्छिक) एक्सेस (Direct access)** — फाइल का कोई भी ब्लॉक उसका नंबर देकर किसी भी क्रम में पढ़ा या लिखा जा सकता है, जैसे किसी अध्याय या डिस्क पर किसी ट्रैक पर कूदना। इसके लिए स्टोरेज का समर्थन आवश्यक है (डिस्क और SSD करते हैं; टेप नहीं) और यह वह प्रतिरूप है जिस पर डेटाबेस जीते हैं: पहले रिकॉर्ड 5,000, फिर रिकॉर्ड 12, फिर फिर से रिकॉर्ड 5,000 — सापेक्ष ब्लॉक नंबर गुणा रिकॉर्ड आकार खोजने की स्थिति देता है।

असली सिस्टम दोनों को एक ही फाइलों पर समर्थन देते हैं (OS या तो प्रत्यक्ष एक्सेस मूल रूप से देता है या उसके ऊपर अनुक्रमिक फाइलों का अनुकरण करता है), और एक तीसरा प्रतिरूप, अनुक्रमित एक्सेस (indexed access), प्रत्यक्ष एक्सेस पर एक अनुक्रमणिका परत चढ़ाता है — अनुक्रमणिका प्रविष्टि खोजिए, उसके सूचक का पालन कीजिए — जिस प्रकार बड़े डेटाबेस रिकॉर्ड शीघ्र खोजते हैं।

अभ्यास सेट 12.2

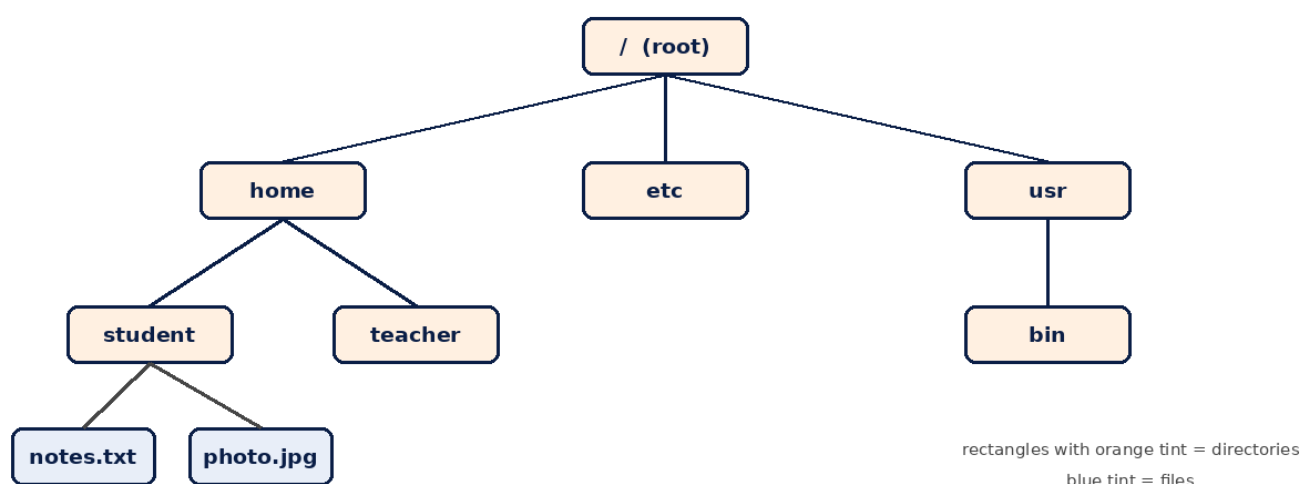
- एक CCTV सिस्टम फुटेज लगातार जोड़ता है और उसे केवल शुरुआत से चलाता है। कौन सी एक्सेस विधि, और एक दिन की रिकॉर्डिंग के दौरान सूचक क्या करता है?
- एक छात्र-रिकॉर्ड डेटाबेस को 10,000 में से रोल नंबर 418 का रिकॉर्ड तुरंत लाना चाहिए। कौन सी विधि, और कौन सी गणना स्थिति खोजती है (निश्चित आकार के रिकॉर्ड)?

12.3 डायरेक्टरी संरचना

फाइलें हज़ारों में बढ़ती हैं; **डायरेक्टरी** वह OS संरचना है जो उन्हें संगठित करती है — खुद एक विशेष फाइल जिसकी प्रविष्टियाँ नामों को फाइलों से जोड़ती हैं। डिज़ाइन, ऐतिहासिक क्रम में:

- **एकल-स्तरीय डायरेक्टरी** — डिस्क की सभी फाइलों के लिए एक डायरेक्टरी (प्रारंभिक सिस्टम, छोटे उपकरण)। सरल, और इसके साथ दो समस्याएँ बढ़ती हैं: हर नाम सभी उपयोगकर्ताओं में अद्वितीय होना चाहिए, और सैकड़ों में से एक फाइल खोजना एक काम है।
- **द्वि-स्तरीय डायरेक्टरी** — एक मास्टर डायरेक्टरी प्रति उपयोगकर्ता एक उपयोगकर्ता डायरेक्टरी रखती है; हर उपयोगकर्ता के पास एक निजी सपाट स्थान होता है। उपयोगकर्ताओं के बीच नाम टकराव समाप्त हो जाते हैं, लेकिन उपयोगकर्ता अपने स्थान के अंदर प्रोजेक्ट संगठित नहीं कर सकते या स्वाभाविक रूप से साझा नहीं कर सकते।
- **वृक्ष-संरचित डायरेक्टरी** — डायरेक्टरी बिना सीमा के उप-डायरेक्टरी रख सकती हैं, वह पदानुक्रम बनाते हुए जो आप हर दिन उपयोग करते हैं: जड़ (/) से एक निरपेक्ष **पथ** किसी भी फाइल को अद्वितीय रूप से नामित करता है — `/home/student/notes.txt` — और हर प्रक्रिया के पास एक **वर्तमान (कार्यशील) डायरेक्टरी** भी होती है ताकि छोटे **सापेक्ष पथ** काम करें (`../teacher` बगल में पहुँचता है)। डायरेक्टरी मिटाना एक नीति प्रश्न उठाता है (यदि खाली न हो तो मना करना, या पुनरावर्ती रूप से मिटाना) जिसका वृक्ष डिज़ाइन को उत्तर देना चाहिए।

A Tree-Structured Directory



चित्र 12.1 - वृक्ष-संरचित डायरेक्टरी: एक जड़, अंदर डायरेक्टरी, पत्तों पर फाइलें।

- **अचक्रीय-ग्राफ़ डायरेक्टरी** — वृक्ष को शिथिल करके एक फाइल/डायरेक्टरी को दो स्थानों पर प्रकट होने दिया गया (एक लिंक या साझा फोल्डर): एक ही सामग्री दो नामों के नीचे, बदलाव दोनों में दिखता है। घेरे ग्राफ़ में फिर भी वर्जित हैं (इसलिए "अचक्रीय"), जो खोजों को सीमित रखता है; लिंक वाले सच्चे सामान्य ग्राफ़ जो घेरे बना सकते हैं, मिटाने पर देखभाल चाहिए (संदर्भ गिनती और कचरा संग्रह) और वही है जो असली सिस्टम सावधानी से लागू करते हैं।

अभ्यास सेट 12.3

- द्वि-स्तरीय सिस्टम में, क्या दो उपयोगकर्ता दोनों `report` नाम की फाइल रख सकते हैं? एकल-स्तरीय सिस्टम में? निर्णायक विशेषता बताइए।
- एक फाइल का निरपेक्ष पथ `/usr/bin/python3` है और कार्यशील डायरेक्टरी `/usr` है। एक सापेक्ष पथ दीजिए, और `..` की भूमिका समझाइए यदि कार्यशील डायरेक्टरी इसके बजाय `/usr/bin` होती।

12.4 फाइल सिस्टम माउंटिंग

अलग डिस्क या विभाजन पर फाइल सिस्टम उपयोगकर्ताओं के लिए अदृश्य होता है जब तक वह **माउंट** न हो: माउंट करना एक फाइल सिस्टम की जड़ को चल रहे सिस्टम के डायरेक्टरी वृक्ष के अंदर एक बिंदु (**माउंट बिंदु**, एक मौजूद डायरेक्टरी) से जोड़ना है, ताकि जुड़े सिस्टम की फाइलें इस प्रकार दिखें जैसे वे एक ही वृक्ष के और फोल्डर हों। पेन ड्राइव लगाना ठीक यही करता है: उसका फाइल सिस्टम `/media/usb` जैसे बिंदु या Windows पर एक ड्राइव अक्षर पर माउंट होता है, और अनमाउंट करना उसे स्वच्छ रूप से

अलग करता है (प्रतीक्षारत लेखनों को फ्लश करने के बाद — "सुरक्षित रूप से हटाएँ" का कारण)। माउंटिंग एक पहेली समझाता है: एक वृक्ष कई उपकरणों में फैल सकता है, क्योंकि वृक्ष OS का तार्किक निर्माण है, किसी एकल डिस्क का नक्शा नहीं।

अभ्यास सेट 12.4

- एक पेन ड्राइव की जड़ में एक फोल्डर `photos` है। `/media/usb` पर माउंट करने के बाद, उस फोल्डर का निरपेक्ष पथ क्या है?
- कोई कॉपी संवाद न दिखते हुए भी, माउंट की गई पेन ड्राइव को अनमाउंट किए बिना खींचना जोखिम क्यों है?

आम गलतियाँ

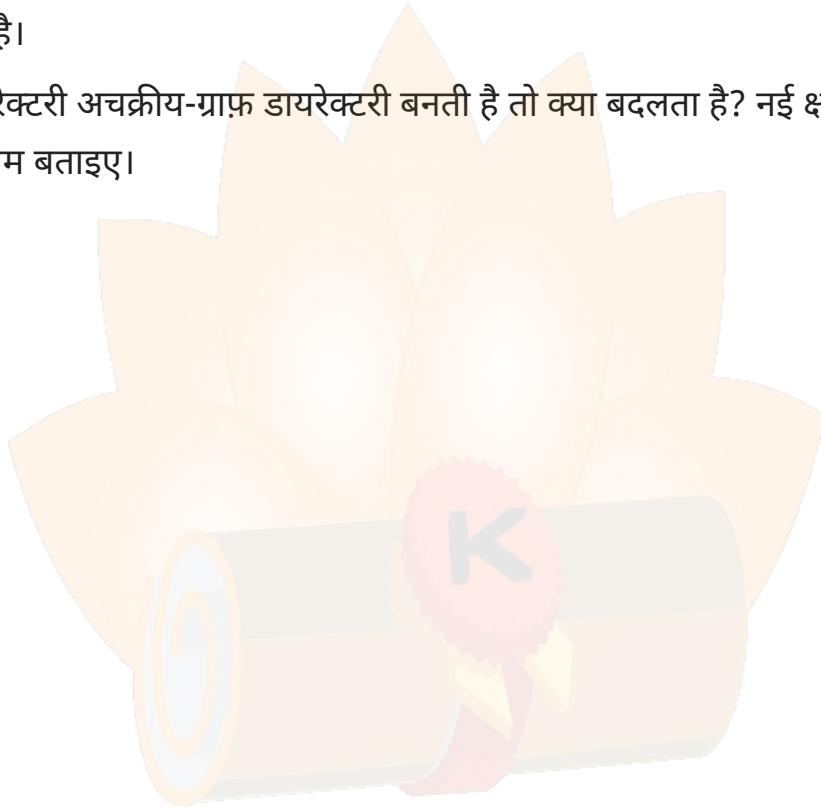
- एक्सटेंशन को सभी सिस्टम के अंतर्गत OS की दृष्टि में फाइल का प्रकार मानना: कुछ सिस्टम प्रकार को आंतरिक रूप से रखते हैं या एक्सटेंशन अनदेखा करते हैं; गुण सूची ही सामान्य उत्तर है।
- प्रत्यक्ष एक्सेस को "तेज़ एक्सेस" कहना। यह क्रम-मुक्त एक्सेस है; उसी हार्डवेयर पर पूरी फाइल का अनुक्रमिक पठन बिखरे प्रत्यक्ष पठनों से तेज़ हो सकता है।
- ऐसे डायरेक्टरी वृक्ष बनाना जो लिंक के द्वारा पूर्वज पर लौटते हैं और फिर भी संरचना को वृक्ष कहना — वह ग्राफ़ है, अपनी विलोपन समस्याओं सहित।

सारांश

- फाइल द्वितीयक स्टोरेज पर एक नामित तार्किक इकाई है, गुणों (नाम, पहचानकर्ता, प्रकार, स्थान, आकार, सुरक्षा, समय-मुहरें, स्वामी) और कार्यों (बनाना, लिखना, पढ़ना, reposition, मिटाना, truncate; open/close से लपेटे) सहित।
- एक्सेस विधियाँ: अनुक्रमिक (क्रम में, सूचक आगे बढ़ता है — टेप, लॉग, मीडिया) और प्रत्यक्ष (ब्लॉक नंबर द्वारा — डिस्क पर डेटाबेस), प्रत्यक्ष पर अनुक्रमित एक्सेस परत सहित।
- डायरेक्टरी विकसित होती हैं: एकल-स्तरीय → द्वि-स्तरीय (प्रति उपयोगकर्ता) → वृक्ष (पथ, कार्यशील डायरेक्टरी) → अचक्रीय ग्राफ़ (लिंक/साझा करना)।
- माउंटिंग एक फाइल सिस्टम को जीवित वृक्ष में एक माउंट बिंदु से जोड़ता है; एक तार्किक वृक्ष कई भौतिक उपकरणों में फैल सकता है।

अध्याय अभ्यास सेट

- फाइल गुणों को गिनाइए और उन दो को चिह्नित कीजिए जिन्हें उपयोगकर्ता सामान्यतः कभी नहीं देखता।
- एक पेट्रोल फाइल हर महीने क्रम में पूरी पढ़ी जाती है; एक ATM शेष फाइल दिन भर खातों के बीच कूदती है। एक्सेस विधियाँ सौंपिए और टेप प्रश्न का न्याय कीजिए: दोनों में से कौन टेप पर रह ही सकती है?
- एकल-स्तरीय, द्वि-स्तरीय और वृक्ष डायरेक्टरी की नाम टकराव और संगठन पर तुलना कीजिए।
- फाइल का निरपेक्ष पथ /home/teacher/marks/cs.xlsx है; कार्यशील डायरेक्टरी /home/student है। इसका सापेक्ष पथ लिखिए।
- माउंटिंग क्या है? एक पेन ड्राइव से समझाइए, जिसमें माउंट बिंदु क्या है और सुरक्षित निष्कासन क्यों मायने रखता है।
- जब वृक्ष डायरेक्टरी अचक्रीय-ग्राफ़ डायरेक्टरी बनती है तो क्या बदलता है? नई क्षमता और नई समस्या का नाम बताइए।





अध्याय 13: फाइल साझा करना और सुरक्षा, फाइल सिस्टम कार्यान्वयन और प्रकार

(BTEUP CSE तीसरा सेमेस्टर · विषय 3.1 · इकाई 5 — फाइल प्रबंधन)

आप क्या सीखेंगे

- फाइलें उपयोगकर्ताओं के बीच कैसे साझा होती हैं, और साझा करना कौन सा संगति प्रश्न उठाता है
- सुरक्षा: एक्सेस प्रकार, एक्सेस कंट्रोल सूची, और स्वामी/समूह/अन्य मॉडल
- फाइल सिस्टम की परतें और वह ऑन-डिस्क संरचनाएँ जो यह रखता है
- डायरेक्टरी कैसे लागू होती हैं, और तीन फाइल आवंटन विधियाँ
- मुक्त स्थान प्रबंधन, कुशलता नोट, और वे फाइल सिस्टम जिनसे आप सच में मिलेंगे

13.1 फाइल साझा करना

साझा करना कई उपयोगकर्ताओं (और एक उपयोगकर्ता की कई प्रक्रियाओं) को एक ही फाइल पर काम करने देता है: एक कक्षा के साझा नोट, एक टीम का प्रोजेक्ट फोल्डर, संस्थान की अंक फाइल। वे तंत्र जो आप अध्याय 12 से जानते हैं — अचक्रीय-ग्राफ़ डायरेक्टरी में लिंक, दूरस्थ स्टोरेज माउंट करने वाले नेटवर्क फाइल सिस्टम — सब इसे देते हैं। साझा करना एक वैचारिक प्रश्न उठाता है जिस पर परीक्षक दबाव डाल सकता है: **संगति (consistency)**। यदि दो उपयोगकर्ता एक साथ संपादित करें, तो किसके बदलाव बचेंगे? सिस्टम **सत्र अर्थविज्ञान (session semantics)** से उत्तर देते हैं (बदलाव तब दिखाई देते हैं जब फाइल बंद होती है — अधिकांश नेटवर्क फाइल सिस्टम का मॉडल) या कठोर **अपरिवर्तनीय साझा**

फाइलों से (साझा फाइल कभी नहीं बदल सकती; अद्यतन एक नया संस्करण बनाते हैं)। एक मशीन पर बहु-उपयोगकर्ता सिस्टम में OS को सुरक्षा भी लागू करनी चाहिए (अगला खंड) ताकि साझा करना स्वामी की पसंद रहे, डिज़ाइन की दुर्घटना नहीं।

अभ्यास सेट 13.1

- दो शिक्षक एक साझा ड्राइव पर एक ही अंक फाइल को एक ही क्षण संपादित करते हैं। सत्र अर्थविज्ञान के अंतर्गत, हर एक दूसरे का काम कब देखता है — और क्या गलत हो सकता है?
- साझा करना सुरक्षा को कम नहीं, अधिक आवश्यक क्यों बनाता है?

13.2 सुरक्षा

जब कोई फाइल बनाई जाती है, तो उसका स्वामी तय करता है कि कौन क्या कर सकता है। OS हर कार्य पर इसे लागू करता है। नियंत्रित किए जा सकने वाले **एक्सेस प्रकार**: read (पढ़ना), write (लिखना), execute (निष्पादित करना), append (जोड़ना), delete (मिटाना), और list (डायरेक्टरी की सामग्री देखना)। तीन कार्यान्वयन दृष्टिकोण:

- **एक्सेस कंट्रोल सूची (ACL)** — हर फाइल एक सूची रखती है: कौन से उपयोगकर्ता/समूह कौन से कार्य कर सकते हैं। सबसे सटीक; सूची लंबी हो सकती है, और उसे जाँचना समय लेता है।
- **स्वामी / समूह / अन्य वर्ग** — संकुचित UNIX योजना: हर फाइल के पास एक स्वामी, एक समूह और शेष संसार होता है; हर वर्ग को एक read/write/execute त्रिक मिलता है (वही rwx जो आपने OS Lab की Linux कमांड में देखा — `ls -l` उन्हें दिखाता है, `chmod` उन्हें बदलता है)। प्रति वर्ग तीन बिट — कुल नौ बिट — फाइल के रिकॉर्ड में समा जाते हैं, गति और सरलता के लिए महीन ब्योरे का व्यापार करते हुए।
- **प्रति फाइल पासवर्ड** — सरल लेकिन अनाड़ी: एक पासवर्ड "वह पढ़ सकती है, वह लिख सकता है" व्यक्त नहीं कर सकता, और पासवर्ड स्मृति से अधिक बढ़ जाते हैं।

सुरक्षा फाइलों से आगे उसे रखने वाली डायरेक्टरी तक फैली है: डायरेक्टरी पर execute (खोज) अनुमति के बिना, उसके अंदर की पठनीय फाइल तक भी नहीं पहुँचा जा सकता — पथ खुद रक्षित है।

अभ्यास सेट 13.2

- `ls -l` एक फाइल के लिए `-rwxr-x---` दिखाता है। कौन क्या कर सकता है? तीनों वर्गों के लिए उत्तर दीजिए।

- ACL और rwx योजना दोनों "केवल स्वामी और उसका समूह इस फाइल को पढ़ सकते हैं" व्यक्त कर सकते हैं। एक नियम दीजिए जो ACL व्यक्त कर सकता है लेकिन rwx नहीं।

13.3 फाइल सिस्टम संरचना और कार्यान्वयन

फाइल सिस्टम खुद परतदार है — हर परत एक काम करती है और नीचे वाली को छिपाती है:

- **एप्लिकेशन प्रोग्राम** फाइल कार्य बुलाते हैं।
- **लॉजिकल फाइल सिस्टम** — नाम, डायरेक्टरी, सुरक्षा और फाइल रिकॉर्ड (FCB) सँभालता है: यह नाम को फाइल की पहचान में सुलझाता है और अधिकार जाँचता है।
- **फाइल-संगठन मॉड्यूल** — फाइल के लॉजिकल ब्लॉक को मुक्त भौतिक ब्लॉक से जोड़ता है (आवंटन, खंड 13.5) और मुक्त स्थान प्रबंधित करता है।
- **बेसिक फाइल सिस्टम** — "भौतिक ब्लॉक N पढ़ो" को ड्राइवर के द्वारा उपकरण कमांड में अनुवादित करता है।
- **I/O नियंत्रण (डिवाइस ड्राइवर + इंटरफ़्ट हैंडलर)** — सच में डेटा ले जाता है (इकाई 4 की दुनिया)।

डिस्क पर, एक विभाजन का फाइल सिस्टम रखता है: एक **बूट ब्लॉक** (OS शुरू करने का कोड, यदि यह विभाजन बूट कर सकता है), एक **वॉल्यूम कंट्रोल ब्लॉक** (विभाजन के योग: ब्लॉक आकार, गिनती, मुक्त-स्थान सूचक), **डायरेक्टरी संरचना**, और प्रति फाइल एक **फाइल कंट्रोल ब्लॉक (FCB)** — अध्याय 12 के गुण और खंड 13.5 के ब्लॉक सूचक; UNIX में FCB प्रसिद्ध inode है। मेमोरी में यह माउंट तालिकाएँ, खुली-फाइल तालिकाएँ और हाल में उपयोग किए गए ब्लॉक के कैश रखता है — यही कारण है कि फाइल का दूसरा खुलना लगभग तुरंत होता है।

अभ्यास सेट 13.3

- कार्य को परत से मिलाइए: (a) जाँचना कि उपयोगकर्ता इस फाइल को लिख सकता है
(b) चुनना कि बढ़ती फाइल को कौन से मुक्त ब्लॉक मिलेंगे
(c) डिस्क हेड को क्रम देना।
- फाइल का आकार डिस्क पर कहाँ रहता है, और फाइल खुली होने पर कहाँ?

13.4 डायरेक्टरी कार्यान्वयन

डायरेक्टरी (नाम → FCB/inode) मैपिंग की एक सूची है; सूची कैसे संगठित है यह खोज गति तय करता है:

- **रैखिक सूची (Linear list)** — प्रविष्टियाँ एक के बाद एक; फाइल बनाना जोड़ता है, खोजना ऊपर से स्कैन करता है — सरल, और डायरेक्टरी हजारों प्रविष्टियों तक बढ़ने पर धीमा।
- **हैश तालिका (Hash table)** — नाम एक स्थिति पर हैश होता है, लगभग तुरंत औसत खोज देता है; टकरावों को शृंखलाबद्ध करना चाहिए, और निश्चित आकार की तालिका अतिप्रवाह हो सकती है — असली सिस्टम अक्सर हैश को छोटी सूचियों के साथ मिलाते हैं।

किसी भी प्रकार, डिज़ाइन लक्ष्य एक ही है: नाम समाधान (अध्याय 12 के पथ) शहर भर की फाइलों वाली डायरेक्टरी में भी तुरंत महसूस होना चाहिए।

अभ्यास सेट 13.4

- एक डायरेक्टरी में 20,000 फाइलें हैं। दुर्भाग्यपूर्ण खोज के लिए रैखिक-सूची तुलनाओं बनाम हैश की तुलना का शब्दों में अनुमान लगाइए।
- डायरेक्टरी डिस्क पर ही क्यों रखी जाती है, जब वह इतनी बार खोजी जाती है?

13.5 फाइल आवंटन विधियाँ — ब्लॉक कहाँ जाते हैं

फाइल का डेटा डिस्क ब्लॉक में रखा जाना चाहिए; **कैसे** रखा जाता है यह गति, वृद्धि और विखंडन तय करता है। तीन क्लासिक विधियाँ (OS Lab का फाइल-आवंटन प्रयोग तीनों को कवर करता है):

सतत आवंटन (Contiguous allocation)। हर फाइल लगातार ब्लॉक की एक धारा घेरती है; उसका रिकॉर्ड केवल प्रारंभ ब्लॉक + लंबाई रखता है। पढ़ना उत्कृष्ट है (भुजा मुश्किल से हिलती है; प्रत्यक्ष एक्सेस एक जोड़ है)। लागत अध्याय 7 की सतत मेमोरी की दर्पण छवि है: डिस्क का बाहरी विखंडन, और फाइल अपनी धारा से आगे नहीं बढ़ सकती — धारा का आकार चुनना वही fit समस्या है, उन्हीं रणनीतियों के साथ।

शृंखलाबद्ध आवंटन (Linked allocation)। हर ब्लॉक फाइल के अगले ब्लॉक के सूचक के साथ समाप्त होता है; रिकॉर्ड केवल पहला ब्लॉक रखता है। कोई बाहरी विखंडन नहीं, फाइलें किसी भी मुक्त ब्लॉक को जोड़कर बढ़ती हैं, अनुक्रमिक पढ़ना स्वाभाविक है। कमज़ोरियाँ: प्रत्यक्ष एक्सेस शृंखला में चलना है (ब्लॉक 500 की लागत 500 पठन), हर ब्लॉक में सूचक स्थान खर्च होता है, और एक खोया सूचक फाइल

का शेष खो देता है। इसकी प्रसिद्ध मरम्मत **FAT** (File Allocation Table) है: सभी सूचक डिस्क की शुरुआत में एक तालिका में चले जाते हैं, जिसे OS मेमोरी में कैश कर सकता है — नीचे के FAT फाइल सिस्टम का डिज़ाइन।

अनुक्रमित आवंटन (Indexed allocation)। फाइल के सभी ब्लॉक सूचक एक **इंडेक्स ब्लॉक** में एकत्र होते हैं (जिसकी ओर फाइल का रिकॉर्ड इशारा करता है)। प्रत्यक्ष एक्सेस अब एक इंडेक्स खोज + डेटा पठन है; कोई बाहरी विखंडन नहीं; फाइलें इंडेक्स भरने तक बढ़ सकती हैं — बड़ी फाइलें इंडेक्स ब्लॉक को शृंखलाबद्ध करती हैं या बहु-स्तरीय इंडेक्स उपयोग करती हैं (UNIX inode ठीक इस मापन के लिए प्रत्यक्ष सूचकों को एकल/दोहरे अप्रत्यक्ष के साथ मिलाता है)। लागत: इंडेक्स ब्लॉक खुद, छोटी फाइलों द्वारा भी चुकाया गया।

| आधार | सतत | शृंखलाबद्ध | अनुक्रमित | |---|---|---|---| | रिकॉर्ड रखता है | प्रारंभ + लंबाई | पहला ब्लॉक | इंडेक्स ब्लॉक नंबर | | अनुक्रमिक एक्सेस | उत्कृष्ट | अच्छा | अच्छा | | प्रत्यक्ष एक्सेस | उत्कृष्ट | बहुत खराब | अच्छा | | बाहरी विखंडन | हाँ | नहीं | नहीं | | फाइल वृद्धि | कठिन | आसान | इंडेक्स आकार तक | | अतिरिक्त स्थान लागत | कोई नहीं | प्रति ब्लॉक सूचक | प्रति फाइल एक इंडेक्स ब्लॉक |

अभ्यास सेट 13.5

- एक वीडियो फाइल रोज़ बढ़ती है और शुरू से अंत तक देखी जाती है। एक डेटाबेस फाइल दिन भर रिकॉर्ड नंबर से पढ़ी जाती है। आवंटन विधियाँ सौंपिए, हर एक का एक कारण।
- FAT में, सूचक कहाँ हैं, और उन्हें हिलाना शृंखलाबद्ध आवंटन की सबसे बुरी कमज़ोरी क्यों बदलता है?

13.6 मुक्त स्थान प्रबंधन

फाइल-संगठन मॉड्यूल को जानना चाहिए कि कौन से ब्लॉक मुक्त हैं। चार योजनाएँ:

- **बिट वेक्टर (बिटमैप)** — प्रति ब्लॉक एक बिट (परिपाटी अनुसार 1 = मुक्त या 0 = मुक्त): संकुचित (4 KB ब्लॉक वाली 1 TB डिस्क को 32 MB बिटमैप चाहिए) और मुक्त ब्लॉक की धाराएँ खोजना आसान बनाता है — सतत आवंटन के लिए आदर्श साथी।
- **शृंखलाबद्ध सूची (Linked list)** — मुक्त ब्लॉक खुद ब्लॉक में रखे सूचकों द्वारा शृंखलाबद्ध: कोई अतिरिक्त स्थान नहीं, लेकिन कई ब्लॉक गिनने या खोजने के लिए भ्रमण धीमा है।
- **समूहीकरण (Grouping)** — पहला मुक्त ब्लॉक कई अन्य मुक्त ब्लॉक के पते रखता है, जिनमें से अंतिम अगले समूह को रखता है: लंबी सूचियाँ टुकड़ों में लाई जाती हैं।

- **गिनती (Counting)** — क्योंकि मुक्त स्थान धाराओं में होता है, (पहला मुक्त ब्लॉक, लगातार मुक्त ब्लॉक की गिनती) जोड़े रखिए: कम प्रविष्टियाँ, धाराएँ ठीक-ठाक वर्णित।

अभ्यास सेट 13.6

- लगभग भरी डिस्क का मुक्त स्थान एकल ब्लॉक का बिखराव है। किस योजना की प्रविष्टि गिनती सबसे कम फटती है, और कौन उसे सबसे अच्छा वर्णित करती है?
- डिस्क भरने पर बिटमैप आकार क्यों नहीं बढ़ता?

13.7 कुशलता, प्रदर्शन, और वे फाइल सिस्टम जिनसे आप मिलेंगे

तीन रोज़मर्रा प्रदर्शन विचार: **ब्लॉक कैशिंग** — हाल में उपयोग किए गए डायरेक्टरी, FAT/इंडेक्स और डेटा ब्लॉक RAM में रखे जाते हैं, अधिकांश पठनों को मेमोरी hit में बदलते हुए; **आगे पढ़ना और लेखन बफ़रिंग** — अनुक्रमिक प्रतिरूप पहले से लाए जाते हैं और लेखन बैच किए जाते हैं; और **ब्लॉक/क्लस्टर आकार चुनाव** — बड़े ब्लॉक बड़ी फाइलों को गति देते हैं लेकिन छोटी पर स्थान बर्बाद करते हैं (आंतरिक विखंडन की फाइल-दुनिया प्रतिध्वनि)। जर्नलिंग फाइल सिस्टम विश्वसनीयता जोड़ते हैं: बदलाव पहले एक लॉग में लिखे जाते हैं, ताकि लेखन के बीच में क्रैश वॉल्यूम को भ्रष्ट करने के बजाय मरम्मत किया जा सके।

| फाइल सिस्टम | आप इससे कहाँ मिलते हैं | एक लाइन में चरित्र | |---|---|---| | FAT32 | पेन ड्राइव, SD कार्ड | सरल, सार्वभौमिक; 4 GB अधिकतम फाइल आकार, कोई सुरक्षा नहीं | | exFAT | बड़ी पेन ड्राइव, कैमरे | 4 GB सीमा के बिना FAT का उत्तराधिकारी | | NTFS | Windows सिस्टम डिस्क | जर्नलिंग, ACL सुरक्षा, एन्क्रिप्शन समर्थन | | ext4 | Linux सिस्टम डिस्क | जर्नलिंग, inode द्वारा अनुक्रमित आवंटन, मज़बूत | | APFS | Apple उपकरण | स्नैपशॉट, एन्क्रिप्शन, copy-on-write डिज़ाइन |

आम गलतियाँ

- आवंटन विधियों की कमज़ोरियाँ मिलाना: सतत → बाहरी विखंडन और वृद्धि; शृंखलाबद्ध → प्रत्यक्ष एक्सेस; अनुक्रमित → इंडेक्स-ब्लॉक लागत। तालिका ही अंक है।
- सुरक्षा को केवल पासवर्ड के रूप में परिभाषित करना। परीक्षा का मूल एक्सेस प्रकार और ACL बनाम स्वामी/समूह/अन्य है।
- शृंखलाबद्ध आवंटन की FAT मरम्मत भूलना — यह इकाई का सबसे अधिक पूछा जाने वाला अनुवर्ती है।
- बिटमैप को बर्बादी कहना: प्रति ब्लॉक 1 बिट पर यह डिस्क की सबसे छोटी संरचनाओं में से एक है।

सारांश

- साझा करना लिंक और नेटवर्क माउंट द्वारा दिया जाता है; सत्र अर्थविज्ञान तय करता है कि संपादन कब दिखाई देते हैं; सुरक्षा साझा करना स्वामी की पसंद रखती है।
- सुरक्षा ACL (सटीक) या स्वामी/समूह/अन्य rwx वर्गों (संकुचित — 9 बिट) के द्वारा read/write/execute/append/delete/list नियंत्रित करती है।
- फाइल सिस्टम परतें: लॉजिकल FS (नाम, FCB, सुरक्षा) → फाइल-संगठन मॉड्यूल (लॉजिकल→भौतिक, मुक्त स्थान) → बेसिक FS → I/O नियंत्रण; डिस्क पर यह बूट और वॉल्यूम कंट्रोल ब्लॉक, डायरेक्टरी और FCB/inode रखता है।
- डायरेक्टरी रैखिक सूचियाँ (सरल, धीमी) या हैश तालिका (तेज़) होती हैं, संभवतः मिली हुई।
- आवंटन: सतत (तेज़, विखंडित, वृद्धि नहीं), शृंखलाबद्ध (बढ़ता है, खराब प्रत्यक्ष एक्सेस; FAT इसे सुधारता है), अनुक्रमित (संतुलित; इंडेक्स-ब्लॉक लागत)। मुक्त स्थान बिटमैप, शृंखलाबद्ध सूची, समूहीकरण या गिनती द्वारा ट्रैक होता है।
- प्रदर्शन कैशिंग, आगे-पढ़ने और ब्लॉक आकार पर टिका है; FAT32/exFAT, NTFS, ext4 और APFS असली जीवन के फाइल सिस्टम हैं।

अध्याय अभ्यास सेट

- FCB को परिभाषित कीजिए और उसकी किन्हीं छह सामग्रियों को गिनाइए।
- बढ़ती लॉग फाइल और निश्चित आकार के डेटाबेस के लिए तीन आवंटन विधियों की तुलना कीजिए — हर बार चुनिए और बचाव कीजिए।
- FAT डिज़ाइन समझाइए: यह शृंखलाबद्ध आवंटन की किस समस्या पर हमला करता है, और कैसे?
- डिस्क पर मुक्त ब्लॉक {2, 3, 4, 9, 15, 16, 17, 30} हैं। मुक्त-स्थान रिकॉर्ड दिखाइए (a) शृंखलाबद्ध सूची (b) गिनती के अंतर्गत।
- `-rw-r-----` की व्याख्या कीजिए और बताइए मध्य त्रिक किस वर्ग का है।
- फाइल सिस्टम की पाँच परतों के नाम बताइए और फाइल-संगठन मॉड्यूल का काम एक लाइन में।
- एक मित्र 6 GB वीडियो ले जाने के लिए 128 GB पेन ड्राइव को FAT32 फॉर्मेट करता है और उसे कॉपी नहीं कर सकता। विफलता और सही फॉर्मेट चुनाव समझाइए।



परिशिष्ट A: ऑपरेटिंग सिस्टम लैब — 12

प्रयोग

(BTEUP CSE तीसरा सेमेस्टर · विषय 3.5 — ऑपरेटिंग सिस्टम लैब। हर प्रयोग उसका उद्देश्य, काम का सार, और वह अध्याय बताता है जिसे वह जीवंत करता है। अपने प्रोग्राम और अवलोकन अपने लैब रिकॉर्ड में इसी क्रम में लिखिए।)

प्रयोग 1 — Linux कमांड। उद्देश्य: शेल से पारंगत होना। फाइल/डायरेक्टरी कमांड (ls, cd, mkdir, cp, mv, rm, cat) का अभ्यास कीजिए, ls -l से अनुमतियाँ देखिए, और chmod से उन्हें बदलिए। जीवंत करता है: अध्याय 2 (सिस्टम प्रोग्राम के रूप में शेल), 12 और 13 (पथ, rwx)।

प्रयोग 2 — Windows स्थापित करना। उद्देश्य: मीडिया से ऑपरेटिंग सिस्टम स्थापित करना। इंस्टॉलर बूट कीजिए, डिस्क विभाजित कीजिए, सेटअप और पहला लॉगिन पूरा कीजिए। नोट कीजिए विभाजन और फॉर्मेटिंग कहाँ आते हैं। जीवंत करता है: अध्याय 10 §10.4 (तीन फॉर्मेटिंग चरण)।

प्रयोग 3 — Task Manager अध्ययन। उद्देश्य: जीवित प्रक्रियाओं का अवलोकन करना। चलती प्रक्रियाओं, उनके PID, अवस्थाओं और CPU/मेमोरी आँकड़ों की सूची बनाइए; एक प्रक्रिया समाप्त कीजिए और सूची देखिए। जीवंत करता है: अध्याय 3 (PCB दृश्य बना)।

प्रयोग 4 — FCFS और SJF शेड्यूलिंग। उद्देश्य: दिए गए प्रक्रिया समूहों पर दोनों एल्गोरिदम का अनुकरण करना और औसत प्रतीक्षा समय की तुलना करना। जीवंत करता है: अध्याय 4 §§4.2–4.3।

प्रयोग 5 — Round Robin और Priority शेड्यूलिंग। उद्देश्य: दोनों का अनुकरण करना; RR में क्वांटम बदलिए और प्रतीक्षा समय व कॉन्टेक्स्ट स्विच पर उसके प्रभाव की रिपोर्ट कीजिए। जीवंत करता है: अध्याय 4 §§4.4–4.5।

प्रयोग 6 — Dining Philosophers। उद्देश्य: मेज़ को इस प्रकार लागू करना कि डेडलॉक हो सके, उसका अवलोकन करना, फिर एक सुधार लागू करना (दोनों स्टिक साथ या अधिकतम चार बैठे) और दिखाना कि डेडलॉक गया। जीवंत करता है: अध्याय 5 §5.5 और 6।

प्रयोग 7 — बैंकर एल्गोरिदम: परिहार। उद्देश्य: सुरक्षा जाँच और माँग-मंजूरी परीक्षण लागू करना; किताब का हल उदाहरण और अपना एक असुरक्षित-अवस्था मामला चलाना। जीवंत करता है: अध्याय 6 §6.4।

प्रयोग 8 — डेडलॉक रोकथाम। उद्देश्य: एक दो-संसाधन परिदृश्य पर एक रोकथाम डिज़ाइन (उदाहरण के लिए, संसाधन क्रम या सब कुछ एक साथ माँगना) प्रदर्शित करना जो उसके बिना डेडलॉक करता है। जीवंत करता है: अध्याय 6 §6.3।

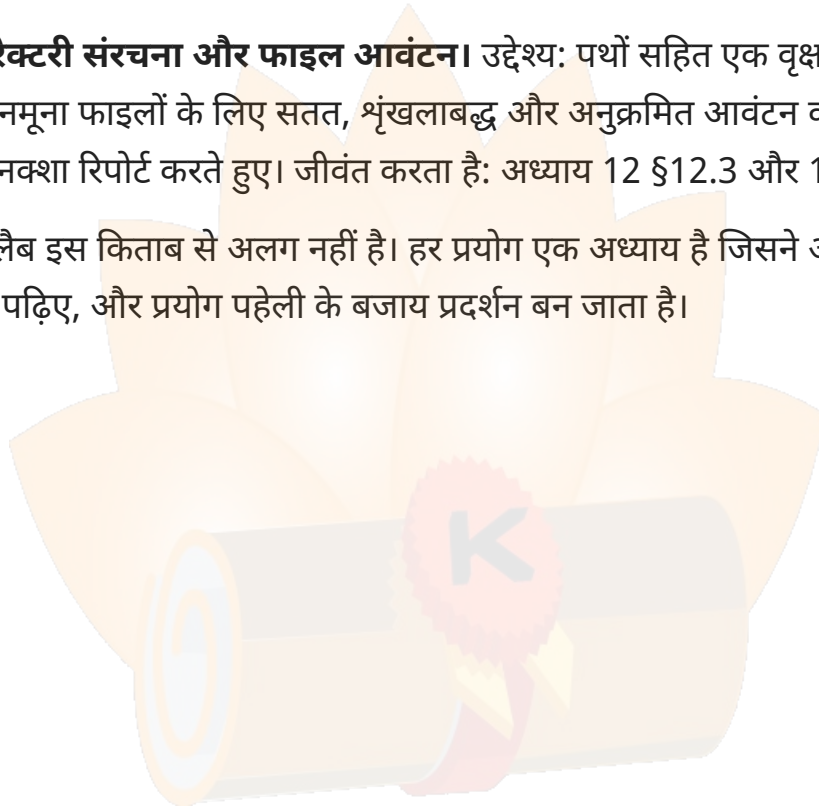
प्रयोग 9 — मेमोरी fit रणनीतियाँ। उद्देश्य: दिए गए छिद्र/माँग सूचियों पर First Fit, Best Fit और Worst Fit लागू करना; अध्याय का हल उदाहरण और उसका सबक पुनः प्रस्तुत करना। जीवंत करता है: अध्याय 7 §7.3।

प्रयोग 10 — पेज प्रतिस्थापन एल्गोरिदम। उद्देश्य: दिए गए संदर्भ स्ट्रिंग पर FIFO, LRU और Optimal लागू करना; FIFO के अंतर्गत Belady की विसंगति दिखाने वाली एक स्ट्रिंग शामिल करना। जीवंत करता है: अध्याय 9 §9.2।

प्रयोग 11 — डिस्क शेड्यूलिंग। उद्देश्य: दिए गए माँग कतारों पर FCFS, SCAN और C-SCAN लागू करना और कुल हेड गति की तुलना करना। जीवंत करता है: अध्याय 11 §11.1।

प्रयोग 12 — डायरेक्टरी संरचना और फाइल आवंटन। उद्देश्य: पथों सहित एक वृक्ष डायरेक्टरी का मॉडल बनाना, और नमूना फाइलों के लिए सतत, शृंखलाबद्ध और अनुक्रमित आवंटन का अनुकरण करना, हर विधि का ब्लॉक नक्शा रिपोर्ट करते हुए। जीवंत करता है: अध्याय 12 §12.3 और 13 §13.5।

एक समापन शब्द: लैब इस किताब से अलग नहीं है। हर प्रयोग एक अध्याय है जिसने अपनी आस्तीन चढ़ाई है — पहले अध्याय पढ़िए, और प्रयोग पहली के बजाय प्रदर्शन बन जाता है।





शब्दावली — इस किताब के मुख्य शब्द

- **Process (प्रक्रिया)** — निष्पादन में चल रहा प्रोग्राम, अपनी मेमोरी और अवस्था सहित।
- **Kernel (कर्नेल)** — OS का हमेशा निवास करने वाला केंद्र, एकमात्र भाग जो हार्डवेयर को सीधे छूता है।
- **System call (सिस्टम कॉल)** — नियंत्रित द्वार जिससे प्रोग्राम कर्नेल सेवाएँ माँगते हैं।
- **PCB** — प्रोसेस कंट्रोल ब्लॉक: एक प्रक्रिया का OS रिकॉर्ड।
- **Context switch (कॉन्टेक्स्ट स्विच)** — एक प्रक्रिया की अवस्था सहेजना और दूसरी की लोड करना ताकि CPU हाथ बदल सके।
- **Critical section (क्रिटिकल सेक्शन)** — प्रक्रिया के कोड का वह भाग जो साझा डेटा छूता है।
- **Semaphore (सेमाफोर)** — केवल atomic wait और signal क्रियाओं से उपयोग होने वाला OS पूर्णांक।
- **Race condition (रेस कंडीशन)** — साझा डेटा एक साथ बदला गया इसलिए परिणाम समय पर निर्भर करता है।
- **Deadlock (डेडलॉक)** — प्रक्रियाओं का समूह जहाँ हर एक वह धारण किए है जिसकी समूह की दूसरी प्रतीक्षा करती है।
- **Safe state (सुरक्षित अवस्था)** — वह अवस्था जहाँ से हर प्रक्रिया फिर भी अपना अधिकतम प्राप्त करके समाप्त हो सकती है।
- **Logical / physical address (तार्किक / भौतिक एड्रेस)** — एड्रेस जैसा प्रोग्राम उसे देखता है / RAM में असली स्थान।
- **Page / frame (पेज / फ्रेम)** — तार्किक / भौतिक मेमोरी के समान निश्चित आकार के ब्लॉक।
- **Page fault (पेज फ़ॉल्ट)** — ऐसे पेज को छूने की घटना जो मेमोरी में नहीं है, इसलिए उसे लाना पड़ता है।
- **Fragmentation (विखंडन)** — अनुपयोगी टुकड़ों में बर्बाद मेमोरी: ब्लॉक के अंदर (आंतरिक) या उनके बीच (बाहरी)।

- **Virtual memory (वर्चुअल मेमोरी)** — RAM से बड़ा प्रति-प्रक्रिया एड्रेस स्पेस, डिमांड पेजिंग द्वारा सेवित।
- **Thrashing (थ्रैशिंग)** — संयुक्त कार्यशील समूह मेमोरी से अधिक होने और फ़ॉल्ट का प्रभुत्व होने से उत्पन्न पतन।
- **Seek time (सीक समय)** — डिस्क भुजा को वांछित ट्रैक तक ले जाने का समय।
- **RAID** — गति (striping) और/या उत्तरजीविता (मिररिंग, पैरिटी) के लिए मिली हुई कई डिस्क।
- **File (फाइल)** — संग्रहीत जानकारी की एक नामित तार्किक इकाई।
- **inode / FCB** — गुण और ब्लॉक सूचक रखने वाला प्रति-फाइल रिकॉर्ड।
- **Mounting (माउंटिंग)** — एक फाइल सिस्टम को जीवित डायरेक्टरी वृक्ष में एक बिंदु पर जोड़ना।

KAMALDNP के बारे में

KAMALDNP वाराणसी, उत्तर प्रदेश में एक IT प्रशिक्षण संस्थान है, जो कंप्यूटर पाठ्यक्रम और ग्रीष्मकालीन प्रशिक्षण देता है, और KAMALDNP.com के द्वारा डिप्लोमा और इंजीनियरिंग छात्रों के लिए अध्ययन सामग्री प्रकाशित करता है।

KAMALDNP · kamaldnp.com · support@kamaldnp.com

