shortest job first (cont.)
guess the future CPU time by the prev. CPU-bound phases using exponential average
$Predicated_{n+1}$
$= \alpha Actual_n + (1-\alpha) Predicted_n$
where α is a weight

shortest remaining time (SRT)
- SJF but use remaining time, is preemptive
+: improved turnaround
-: can cause starvation (small tasks arriving will push longer tasks back)

interactive environments
- *response time*: time between request and response by system
- *predictability*: variation in response time, lesser variation

use preemptive scheduling algorithms → scheduler runs periodically
- use a timer interrupt to invoke scheduler

time quantum: execution duration given to process
- could be constant or variable among processes, must be multiples of interval or timer interrupt

round robin
- tasks stored in FIFO queue, pick first task from queue to run until:
  - fixed time quantum elapsed, task gives up CPU voluntarily, task blocks
- task is placed at end of queue to wait again
  - blocked task will be moved to another queue to wait for its request

+: response time guarantee - given n tasks and quantum q, time before CPU is given is bounded by $(n-1)q$
+: fair alloc of CPU time
-: performance depends on time quantum

priority scheduling
each process is assigned a priority, lower integers indicate a higher priority

non-preemptive: once process starts executing, continues until completion/relinquish CPU
preemptive: running process can be interrupted if higher priority process enters the queue

+: efficient handling of critical tasks
-: starvation for lower prio task esp. preemptive
-: priority inversion (for critical sections)

Multi-Level Feedback Queue (MLFQ)
- adaptive, minimizes both: response time for IO bound processes and turnaround time for CPU bound processes

basic rules:
1. if $Prio(A) > Prio(B)$ → A runs
2. if $Prio(A) == Prio(B)$ → A and B run in RR

priority setting:
1. new job → highest priority
2. if a job fully utilized its time slice → priority ↓
3. if a job blocks before it finishes time slice → priority retained

-: gaming the system (relinquish CPU just before time slice ends to maintain priority)
-: I/O bound starvation (many short-running CPU jobs may continuously occupy highest priority queue, starving I/O bound jobs that do not need CPU time but need quick responses)
-: priority inversion

lottery scheduling
- give out "lottery tickets" to processes for various system resources
  - lottery ticket chosen randomly among eligible tickets
  - winner granted the resource
- in the long run, process holding X% of tickets wins X% of resource

+: starvation prevention, every process has a chance to be selected
+: fair
-: overhead, probabilistic nature, lack of priority

threads
when there's an expensive process like:
- process creation in `fork()` - duplicate mem. space & process context
- context switch - saving of process info

single process can be multithreaded
- identification (usually **thread id**)
- registers (general purpose and special)
- "stack"

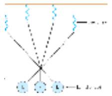
threads ensures **economic handling** of processes, **resource sharing**, **responsiveness** of the program and **scalability**

-: system call concurrency (parallel execution of multiple threads)
-: process behaviour: impact on process operations
`fork()`:
only duplicates the calling thread in child process, other threads in parent not copied to child, issues if other threads held resources
`exit()`:
if any thread calls exit(), entire process terminates, all threads end immediately, no chance for other threads to clean up
`exec()`:
replaces entire process, including all threads, new program starts with single thread, all previous threads terminated, `exec()` doesn't return if successful

user thread
thread implemented as a user library
- a runtime system will handle thread op
- kernel unaware of threads
+: can have multithread on any OS
+: just library calls, more flexible
-: OS is not aware of threads, scheduling performed at process level (cannot exploit multiple CPUs, one thread blocked → all threads blocked due to process block)

kernel thread
thread is implemented in the OS
- thread op. is handled as system calls
- thread-lvl scheduling possible
+: kernel can schedule on thread level (multiple CPUs can be used)
-: thread ops on a system call (slow)
-: generally less flexible

hybrid thread



uses both kernel and user threads
- OS schedule on kernel threads only, user thread binds to a kernel
+: offer great flexibility, can limit concurrency of any user

POSIX threads
`pthread_t:` thread id
`pthread_attr:` attributes of a thread
`pthread_exit(): void`
`pthread_exit(void *retval)`
termination when `start_routine` ends
`pthread_join():int`
`pthread_join(pthread_t thread, void **retval)`
waits for specified thread to terminate

memory sharing
-threads within a process share a memory space
-global variables are accessible by all threads

inter-process communication
1. shared memory
- process P_1 creates a shared memory region M
- process P_2 attaches memory region M to its own memory space
- P_1 and P_2 can now communicate using M
  - M behaves similar to a normal mem. reg.
+: efficient, easy to use
-: synchronization, harder implementation

shared mem creation: `shmget` in the master program
attachment: use `shmat` in both programs
control flag: use `shm` as a synchronization mechanism
data passing: write data in slave (`shm`[1..3]), read in master
cleanup: master detaches (`shmdt`) and destroys (`shmctl`) the shared memory

2. message passing
- process P_1 prepares a message M and sends it to P_2
- process P_2 receives message M
- send and recv. msgs are provided as system calls
- properties: **naming** (how to identify the other party in comms) + **synchronization** (behaviour of sending/receiving operations)
- the `Msg` has to be stored in kernel memory

naming scheme: direct communication
- sender/recv.er of message explicitly names other party
- one link per pair of communicating messages, need to know the identity of the other party

naming scheme: indirect comms.
- message sent to/recv.ed from mailbox/port
- one mailbox shared among a number of processes

synchronization
- blocking primitives (synchronous)
- non-blocking primitives (asynchronous)

+: portable, easier synchronization
-: inefficient, harder to use

UNIX pipes
| : piping - links input/output channels of one process to another
- pipe can be shared between two processes (producer-consumer relationship)
- **behavior:** unidirectional data channel for interprocess communication
  - producer writes to one end, consumer reads from the other end
  - blocking operations: read() blocks on empty pipe, write() blocks on full pipe

- **semantic**: pipe functions as circular bounded byte buffer with implicit synchronization
  - writers wait when buffer is full
  - readers wait when buffer is empty

- **variants:**
1. multiple readers/writers
2. half-duplex/unidirectional: one write end and one read end
3. full-duplex/bidirectional : both ends read & write

`pipe()` function creates an interprocess communication channel
- returns an array of two file descriptors:
`fd[0]`: Reading end of the pipe
`fd[1]`: Writing end of the pipe
syntax: `int pipe(int fd[])`

signals
- are asynchronous notifications sent to processes or threads to inform them of specific events.

- default handlers (e.g., terminate, stop, etc.)
- user-defined handlers for certain signals

common signals in Unix:
- `SIGKILL:` immediate termination (cannot be caught or ignored)
- `SIGTERM:` graceful termination (can be handled)
- `SIGINT:` interrupt (e.g., Ctrl+C)
- `SIGSTOP/SIGCONT:` stop and continue process execution
- `SIGSEGV:` segmentation fault
- `SIGFPE:` arithmetic errors (e.g., divide by zero)

signals are lightweight but limited IPC mechanisms, often used for event notifications or process control.
- they can be sent using system calls like `kill()` or generated by the kernel in response to events such as hardware interrupts or memory violations.

synchronization
- problem: two or more processes execute concurrently in interleaving fashion and share a modifiable resource → synchronization problems
- executing concurrent processes may be non-deterministic
- outcome depends on the order in which the shared resource is accessed/modified → **race conditions**

P_1 and P_2 share variable x
- $X = X + 1000$ translates to roughly:
1. $Load\ X \rightarrow Register1$
2. $Add\ 1000\ to\ Register1$
3. $Store\ Register1 \rightarrow X$

good behaviour race conditions means that the instruction calls by P_1 and P_2 do not overlap

critical section
- designate code segment with race condition as critical section
- only **one process** can execute in the critical section

good CS implementation
1. **mutual exclusion:** if P_1 is executing in crit. sec., all other P are prevented from entering crit. sec.
2. **progress:** if no process is in a crit. sec., one of the waiting processes should be granted access
3. **bounded wait:** after P_i requests to enter crit. sec., there exists an upperbounded of number of times other processes can enter the crit. sec. before P_i
4. **independence:** process not executing in crit. sec. should never block other process

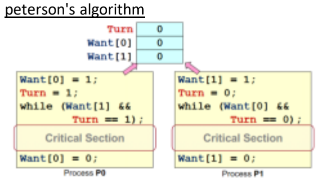
incorrect synchronization
1. **deadlock:** all processes blocked
2. **livelock:** usually related to deadlock avoidance mechanism
- processes keep changing state to avoid deadlock and make no other progress
3. **starvation:** some processes are blocked forever

test and set
- machine instruction provided by processor to aid synchronization
`TestAndSet Register, MemLocation`
- behaviour:
1. load the current content at MemLocation into Register
2. stores a 1 into MemLocation

- the above is performed as a single machine operation *(atomic)*

busy waiting
- keep checking the condition until it is safe to enter crit. sec. → waste of processing power

peterson's algorithm



- assume writing to `Turn` is an atomic operation

peterson's algo (cont.)
-: busy waiting → waiting process repeatedly test the while-loop condition instead of going into blocked state

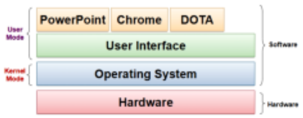
-: low level: HLL programming construct is desirable
- simplify mutual exclusion
- less error prone

-: not general
- general sync. mechanism is desirable (not just mutual exclusion)

OS: a program that acts as an intermediary between a computer user and the computer hardware
- batch OS: execute user program one job at a time
- OS is a resource allocator which arbitrates conflicting requests for efficient and fair resource use
- also known as the kernel
- deals with hardware issues
- provides system call interface
- special code for interrupt handlers, device drivers

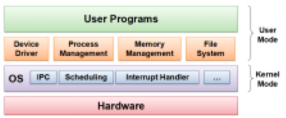
<u>time-sharing OS</u>
OS provides sharing of CPU time, memory and storage

<u>modern OS</u>
- PC, mobile, real-time, embedded



<u>OS structures</u>
1. monolithic OS
- kernel is: one big special program
- OS handles process management, memory management, file system and offers system call interface to programs + device drivers to hardware

2. microkernel OS
- kernel is: small and clean, provides basic and essential facilities, Inter-Process Communication (IPC)



3. virtual machine
- software emulation of hardware, virtualization of underlying hardware

process management
- to be able to switch from prog. A to prog. B requires:
  1. info regarding execution of program A
  2. program A's information is replaced with information to run program B
- process: an abstraction to describe a running program

<u>process abstraction</u>
- process is a *dynamic abstraction* for executing program
- information required to describe a running program

components
- memory: storage for instr. and data
- cache: duplicate part of mem. for faster access + split into instr. and data cache

components (cont.)
- **fetch unit:** loads instr. from memory, location indicated by Program Counter
- **functional units:** carry out the instr. execution, dedicated to different instr. type
- registers: internal storage for fastest access
  - **General Purpose Register (GPR):** accessible by user program (visible to compiler)
  - **Special Register:** PC, Stack Pointer, Frame Pointer, Program Status Word etc.

<u>basic instruction execution</u>
instr X is fetched (using PC)
instr X dispatched to corresponding Functional Unit
- read operands if applicable from memory or GPR
- result computed
- write value if applicable (usually to memory or GPR)
instr X is completed (PC updated for next instr.)

<u>stack memory</u>
- new memory region to store information for function invocation
- **stack pointer**: top of stack region
- information is described by a stack frame
- **stack frame:**
  - return address of the caller
  - arguments for the function
  - storage for local variables

<u>setup of stack frame</u>
- caller: pass params with registers and/or stack
- caller: save return PC on stack
- transfer control from caller to callee
- callee: save the old Stack Pointer (SP) and FP, save registers used by callee
- callee: allocate space for local vars of callee on stack
- callee: adjust SP to point to new stack top

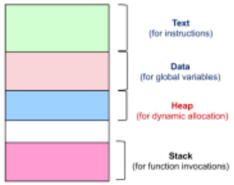
<u>teardown of stack frame</u>
on returning from function call:
- callee: restore saved Stack Pointer, FP and saved registers
- transfer control back to caller using saved PC
- caller: continues execution in caller

frame pointer (platform dependent)
- facilitate access of various stack frame items
- FP points to a fixed location in a stack frame
  - other items are accessed as a displacement from the FP

saved registers
- GPRs are limited, when exhausted:
  - use memory to temporarily hold GPR value
  - GPR can then be reused for other purpose
  - GPR value can be restored after from mem.

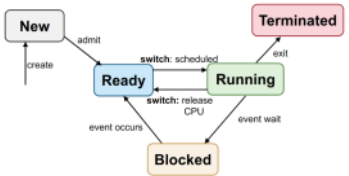
known as register spilling

<u>dynamically allocated memory</u>
- in C, `malloc()`
- allocated only at runtime → cannot place in Data
- no definite deallocation timing → cannot place in Stack



process identification
- distinguish processes from each other

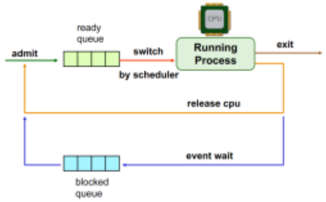
process state
- indicates execution status



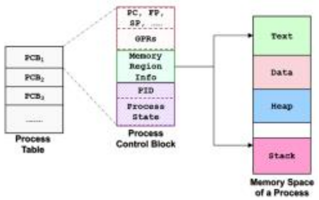
blocked: process waiting for event, cannot execute
switch (running→ready): process gives up CPU voluntarily or preempted by scheduler
event wait(running→blocked): process requests resource which is not available (e.g. system call/waiting for I/O)

<u>global view of process states</u>
- given n processes:

1 CPU	*m CPUs*
≤ 1 process in running state, conceptually 1 transition at a time	≤ m process in running state, possibly parallel transitions



process table & process control block



system calls
- API to OS: way of calling facilities in kernel
- have to change from user mode to kernel model
- Unix system calls in C/C++:
  - library version with same name and params: lib. ver. acts as function wrapper
  - user friendly library version: lib. ver. acts as function adapter

<u>general system call mechanism</u>
1. user program invokes library call
2. library call places the **system call number** in a designated location (e.g. Register)
3. library call executes a special instruction to switch from user mode to kernel mode
4. now in kernel mode, the appropriate system call handler is determined:
- handled by dispatcher
5. system call handler is executed
6. system call handler ended:
- control return to library call
- switch from kernel mode to user mode
7. library call return to the user program
- via normal function return mechanism

<u>exception</u>
- exception is synchronous (occurs due to program execution)
- exception handler is executed automatically

<u>interrupt</u>
- external events can interrupt the execution of a program
- interrupt is asynchronous (events that occur independent of program execution)

<u>exception/interrupt</u>
1. when exception/interrupt occurs
- control transfer to a handler routine automatically

2. return from handler routine
- program execution resume, may behave as if nothing happened

<u>process abstraction in Unix</u>
- identification: PID
- information:
  - process state: running, sleeping, stopped, zombie [a <u>process</u> that has completed execution (via the <u>exit</u> <u>system call</u>) but still has an entry in the <u>process table</u>: it is a process in the "<u>terminated state</u>"]
  - parent PID
  - cumulative CPU time (total CPU time)

<u>process creation in Unix: `fork()`</u>
- creates a new process (the **child process**)
- child process is a duplicate of the current executable image
  - same code, same address space etc.
  - data in child is a copy of the parent (not shared)
- child differs from parent in terms of:
  1. PID
  2. parent
  3. `fork()` return value

both parent and child processes continue executing after `fork()`

common usage is to use parent/child processes differently - use the return value of `fork()` to distinguish parent and child

when `fork()` is executed, it creates a new child that is an exact process, **including memory space (meaning all local vars will be cloned)**

<u>`execl()`</u>
to replace current executing process image with another one
- code is replaced
- **BUT PID and other information still intact!**

Syntax
```
int execl( const char *path,
           const char *arg0,
           ...,
           const char *argN, NULL );
```

- **path**: Location of the executable
- **arg0**, ..., **argN**: Command Line Argument(s)
- **NULL**: To indicate end of argument list

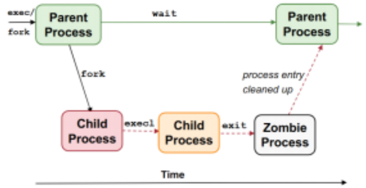
<u>`init` process</u>
- created in kernel at boot up time
- has a PID = 1

<u>`void exit(int status)`</u>
- to end execution of process, status is returned to parent process
- 0 = normal termination, !0 = indicate problematic execution
- on exit: most system resources used by process are released on exit
  - however, some process resources are not releasable:
    1. PID & status needed (for parent-child synchronization)
    2. process accounting info e.g. CPU time
    → process table entry may also still be needed

<u>`int wait(int *status)`</u>
parent process can wait for child process to terminate
- returns PID of the terminated child process
- use `NULL` if info is not needed

behaviour of `wait`
- this call is blocking (parent process blocks until at least one child terminates)
- call cleans up remainder of child system resources (i.e. those not removed on `exit()`)
- hence, this kills zombies!

`waitpid()` - waits for a specific child process
`waitid()` - waits for any child process to change status

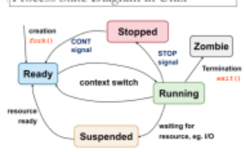


`wait()` "creates" zombies
- child on exit becomes zombie
  - cannot delete all process info
  - cannot kill zombie (since process is already dead)

zombie cases!

parent process terminates before child process	child process terminates before parent but did not call `wait`
- `init` process becomes pseudo parent of child process - child termination sends signal to `init` which calls `wait()` to cleanup	- child process becomes a zombie process - can fill up process table as a result - may need a reboot to clear the table

Process State Diagram in Unix



<u>`fork()` implementation</u>
1. create address space of child process
2. allocate `p' = new PID`
3. create kernel entry in process table
4. copy kernel environment of parent process (e.g. priority for process scheduling)
5. initialize child process context (`PID = p'`, `PPID = parent id`, `CPU time = 0`)
6. copy memory regions from parent
- program, data, stack (expensive op)
7. acquire shared resources
8. initialize hardware context for child process: copy registers from parents proc.
9. child process is now ready to run (add to scheduler queue)

<u>process scheduling</u>
1. **batch processing:** no user, no responsiveness
2. **interactive:** active user, responsive, consistent response time
3. **real time:** deadline to meet, periodic process

criteria for all processing environments
- fairness: should get a fair share of CPU time
  - per user/per process basis, no starvation*
- balance: all parts of computing should be used

*starvation: process is perpetually denied access to necessary resources

scheduling policies
1. non-preemptive
process stays scheduled until: it blocks/relinquishes CPU time voluntarily

2. preemptive
process is given a fixed time quota to run, at the end of the time quota: another process gets picked if avail.

scheduling a process
scheduler triggered (OS takes over) → context switch if needed → pick a suitable process P based on scheduling algo → setup context for P → let process P run

batch processing
- *turnaround time*: total time taken [$finish - start\ time$], related to time waiting for CPU
- *throughput*: number of tasks finished/unit time
- *CPU utilization*: % of time when CPU is working on a task

<u>FCFS</u>
+: guaranteed to have no starvation
-: simple reordering can reduce average waiting
-: convoy effect (first task is CPU bound, followed by a number of IO bound tasks which causes CPU idle)

<u>Shortest Job First (SJF)</u>
- smallest total CPU time
- need to know total CPU time in advance
+: minimizes average waiting time