Lecture 4

Process internal structure

Last week we understand that virtual machines, or processes, are implemented using dual mode rather than by interpreting programs.

This week we will see what data the OS must keep for each process to implement a virtual machine.

References:

- [OSC] Sections 4.1–4.3 (Process), 5.1, 5.2, 5.4, 5.5, 5.7 (Threads), 9.3, 9.4 (Paging)
- [ULK] pp. 45–62 (Paging in Hardware/Linux), 65–82 (Process Descriptor), 198–205 (Memory descriptor/region), 233–236 (System calls).

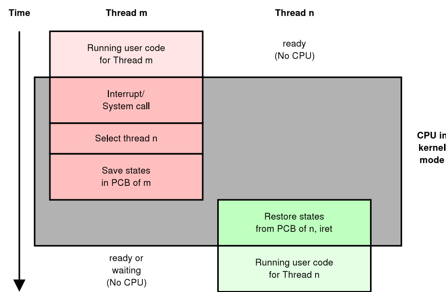
What a process includes

The kernel use a **Process Control Block** (PCB) to keep track of the **resource allocated to the process**:

- **Scheduling information**: how much CPU time it has used, how much it has used recently, whether it is ready for execution, etc.
- **Memory resource**: which part of physical memory is allocated to the process, how the physical memory is to be translated to virtual memory.
  Include data, code, stack, etc., but not the registers controlling them.
- **Context information**: what is the values of the registers when the CPU executes the process for the last time.
  Include general registers and special ones, e.g. instr. and stack pointers (IP, SP).
- **Kernel stack**: for use by kernel after mode switch.
- **File resource**: what files had the process opened, what is the current status of those files (e.g., access pointers), etc.

Context switching

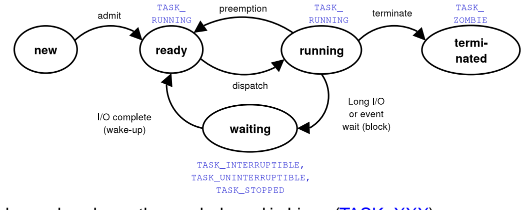
When the OS decides to switch to run another process, the **context** of the related processes are **saved** and **restored**:



This is called a **context switch**.

Process states

Each process has a **state**, denoting whether it is ready for execution, and if not, why. E.g., it might be waiting for I/O to complete.



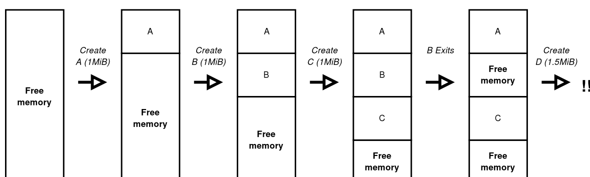
The above also shows the symbol used in Linux (TASK_XXX).

A **ready** process can be put onto a CPU, while a **waiting** one cannot. The latter are typically **waiting for some events** to occur.
In Linux, "ps l" gives you some idea about what processes are waiting for.

What about the memory?

Before we actually examine how the virtual memory is organized, let's look at an organization that does **not work**.

Suppose every process requests a block of contiguous memory that is large enough…



We call this problem **external fragmentation**. ("External" refers to that no running process is responsible for the wasted memory.)
Problem: we can't use memory holes left behind when a process **completes**.

How to deal with the problem? Some tries…

- Soln 1: Find a good **allocation strategy**: **Best/Worst/First fit**: always use the smallest/largest/first hole that fits .
- But… it **doesn't solve the problem**. Best/First fit is better than worst fit, but simulations show that it still waste 33% space.
- Soln 2: **Force every program to be the same size**. Then no memory wasted on the holes.
- But… Even small program uses as much memory as large programs! **Internal fragmentation**. (Early IBM 360 uses such strategy.)
- Soln 3: **Compaction**. Eliminate holes by moving programs around.
- But… it is **slow** to move big memory. When memory exhaustion is close, compaction is done frequently: **serious performance degradation**.
  The strategy is used by IBM 370 and Palm. If you use a Palm, you will notice that it seems suspended from time to time to do compaction.

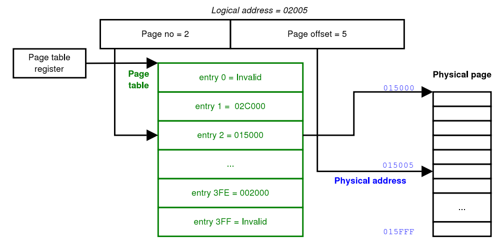
Better solution: fixed-sized, small memory blocks

A more radical solution: don't request for contiguous physical memory!

- Memory is **partitioned** into **fixed-size** blocks, or **frames**.
  How small? Typically, from 1KiB to 16KiB, depending on processors. Ki means "kibbi" or "kilo-binary", which means 2^{10}.
- Let's make it concrete. Suppose both logical and physical addresses are 22-bit long, allowing 4MiB of memory. Each frame is of 4KiB.
- The physical memory is divided into 4MiB/4KiB = 1024 frames.
- Each process would have a **page table** of 1024 entries. The page table itself requires around 4KiB, so the OS gives 1 frame to it.
- The process can ask the OS: "give me 16KiB memory with virtual addresses from hex 2000 to 5fff". The OS gives the process 4 frames, and write their physical addresses to entries 2–5 of the page table.

How to use the memory?

Suppose we want to access the virtual memory at hex address 002005…



So the logical address is split into a **page number** and **page offset**.

The page number is used to lookup the physical frame, the page offset is then added to it to find the actual physical address of the memory needed.

Problem solved?

- **External fragmentation**? No—frames have the same sizes.
- **Internal fragmentation**? Yes, but its small enough (average 2KiB per allocation) to tolerate.
- **Waste some space for the page table**, okay? Hmm… sounds reasonable, each process needs only 4KiB.
- But… programs are larger and slower, since every memory access must actually be 2 memory accesses, right?
- Larger: No. Everything is done by **hardware** for all modern processors.
  There is a "paging mode". When enabled, all memory access (including code access) will be translated by the Memory Management Unit (MMU).
- Slower: yes (unluckily), but most of the time we look at the same page table entries. A **cache** of page table entries reduces the loss to 1-5%.

Side effects: protection, shared memory

- Changing the mapping is a **privileged** operation.
  And the processes cannot directly access physical memory.
- Processes simply has **no way to access memory that does not belong to it**: no logical memory is associated with those physical memory.
- It is easy to implement **shared memory**: we just have two process sharing some memory frames.
- Note that the lower ordered bits of each page table entry we have mentioned are **always 0**.
- Instead of really storing 0's there, most CPUs use them to keep **information about the entry**, e.g., whether the table entry is invalid.
- Another common use: to mark some page to be **unwritable**.
  There are more bits which we will see when we talk about swapping.

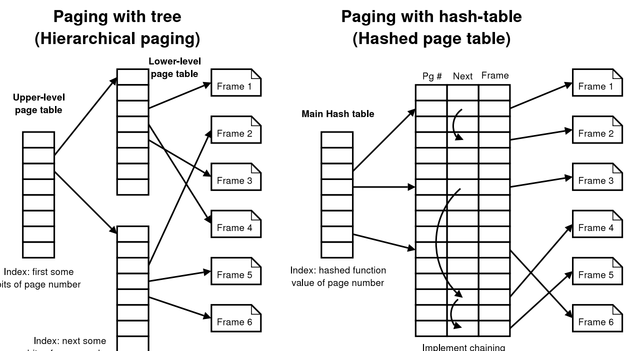
Larger address space

- In our example, we talk about 22-bit addresses, which gives 4MiB memory. This is very small in today's standard.
- What will happen if we have 32-bit addresses?

  a. Make the **page** larger, resulting in larger internal fragmentation.

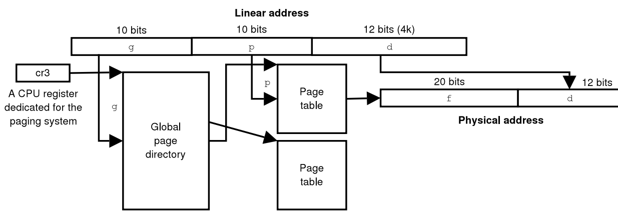
  b. Make the **page table** larger, wasting more space on page tables.
- These penalties are there **even for small programs**.
  So it's a real problem: what if we run 300 small programs in a large computer?
- **Solution**: We can consider the page table to be a **data structure** to **search** a frame given a virtual page number.
- Then we can use data structure techniques to deal with the problem.
  The data structure may be supported by hardware or by OS. In the latter case, during a page table cache miss, the OS is asked to find the frame address.

What data structure to use?



Cost: more levels or indirections will make memory access slower.
We can always increase the page table cache size, however.

Hardware to do hierarchical paging: 80386–Pentium-IV



- **Large (4GiB) address space**: a two-level paging scheme is used.

  Pentium-Pro may has 16GiB address space. In such case it requires a 3-level tree structure.
- To **reduce the number of page tables**, a global page directory entry may point to a 4MiB page, skipping the page table.
- For architectures with 64-bit addresses, paging with trees is not good.

Data to store in the PCB

- The **physical address** of the page table.

  This address is put into the page table register during a context switch.
- The **ranges of virtual addresses** that has been allocated to the process, and the permissions requested.

  Does it duplicate the information in the page table? We will later see that it doesn't… but hold on.
- How about the page table itself? It might be contained within the PCB, but most OS keep it a separated part.

  This allows multiple processes to share the same virtual memory, as we will examine next.
- **Aggregate information** about the memory allocated. E.g., total size.

  This is for profiling and debugging and for limiting the maximum size allocated.

Multiple threads in a process

- So far, we assume that the virtual machine has exactly 1 "virtual CPU", so at any time one part of the program is being executed.
- Sometimes we want a virtual machine with **multiple virtual CPUs**, so that at any time multiple parts of the program is executed.

  If there is only 1 physical CPU, one can time-share it as before.
- Why we want it?
  1. **Make coding easier**. E.g., our program have a part waiting for user input and another part doing background computation.
  2. **Use multiple real CPUs**. We hope the virtual CPUs will map to many real CPUs so that our computation is done faster.
- Note that our program will have more than one place executing at the same time. The *code path* followed by each CPU is called a **thread**.

Benefit of threads

Why not use **multiple processes**, but instead have to use multiple virtual processors and threads within the same process?

- **Save resources**: allocate only 1 set rather than many sets.
- **Cheaper to do context switch**: e.g., the memory does not have to switch, so the processor cache need not be flushed.

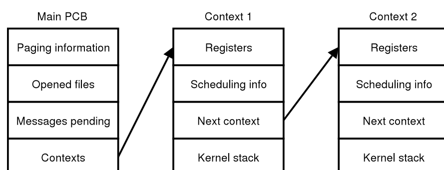
  So threads are usually called Light Weight Processes (LWPs).
- **Easier communication among threads**: the whole memory image is shared between the threads, and the threads uses the same address to access the same memory.

  Shared memory requested by different processes in general may have different virtual addresses, so pointers will fail.

Downside: a lot more chance for threads to interfere each other.

Implementation

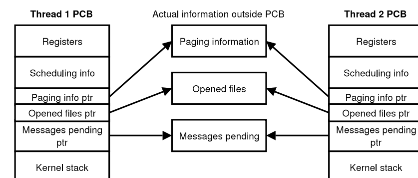
- How to give multiple contexts to a process?
- Easy: we can just give a **linked list of context** for each process…



- The ready process list will become ready thread lists, so that the scheduler will choose a ready thread to run rather than a ready process.
- This structure is used by most Unix.

Alternative implementation

Linux uses a different design: it gives a PCB to each thread, and allow different threads to **share** information like page table and opened files .



It seems like each thread is a virtual machine, but two virtual machines can use exactly the same memory and opened files if they want.

- Linux calls threads to be "tasks", and processes to be "task groups".
- More **flexible**: threads can share some but not all resources.

  E.g., they can share the memory but not the files ,or vice-versa.

Programming interface

- Of course, we need C functions to allow us to create, delete, and manipulate threads.
- Each OS provides a **different system call** for creating threads:
  - Solaris: use *lwp_create*() to create a new "light weight process" within the same process.
  - Linux: use *clone*() (similar to *fork*()) to create a new task, with some resources shared with the old process.
- This is no good for **interoperability**. So a POSIX threads (PThread) library standard is established.
- A *pthread_create*() call is used to create a new thread, with parameters controlled by an argument of type *pthread_attr*.
- All major Unix and Windows-NT/2000 supports PThreads.

Example: a simple PThread program

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
void *func(void *arg) {
    int i = (int)arg;
    printf("In thread %d.\n", i); fflush(stdout);
    sleep(i);
    return NULL;
}
int main() {
    pthread_t t[5];
    void *val; int i;
    for (i = 0; i < 5; ++i)
        pthread_create(&t[i], NULL, func, (void *)i);
    for (i = 0; i < 5; ++i) {
        pthread_join(t[i], &val);          // similar to waitpid()
        printf("Done waiting for %d.\n", i); fflush(stdout);
    }
}
```


User-mode threads

- How costly is it to create a thread? It depends on the OS.
  Linux: quite cheap; most Unix: expensive. Making many (e.g., 1000) threads is usually bad, although someone show it possible to create 1 million in Linux 2.5.
- Sometimes multithreading is used to **exploit multiple processors**. In such cases such **"kernel-based" threads** must be used.
- At other times, multithreading is used just for **ease of coding**. Then it is possible to do "multithreading" without telling the kernel.
- The program would allocate some memory to hold **a context and a stack**, and do context switch among threads by itself.
- The program must not use functions that let the OS **block the process**, or **all threads will stop**. E.g., if a thread reads a file ,it must first make sure there is **already** something to read.
- Some pre-made libraries do all these for you (e.g., GNU pth).

Threading models

- Benefit of kernel-based threads: can use multiple processors.
  Programs can also make blocking calls at will, but let's not call it a benefit.
- Benefit of user-mode threads: cheap, thus can create many threads.
  Why cheaper? Context switch does not involve the kernel—no mode switch!
- It is possible to **combine some of the advantages**: the program can create a few kernel-based threads to "pick-up" a large number of user-mode threads.
- Such libraries are said to use "**Many-to-Many Model**".
  Meaning many user-mode threads map to many kernel-based threads.
- Kernel-based threads and user-mode threads are said to be **One-to-One** and **Many-to-One** respectively.
- Many-to-many model might seem the best approach, but it is complicated to write the library and more difficult to make it really fast.