CS2106 Live Class Lec2a: Overview of process management

$_{/}|/|U_{C}h@NgRu!$

May 10, 2021

1 The focus area of the module

1. OS structure and architecture
2. Process management
3. Memory management
4. File management
5. OS protection mechanism

2 Introduction to process management

As the OS, to switch from running program A to program B requires:

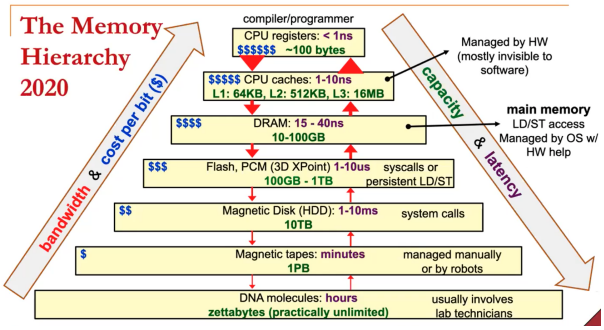
2.1 information regarding the execution of program A needs to be saved somewhere

2.2 Program S's information is replaced with the information required to run program B

Hence a concepts of process is needed
process is not the program but the dynamic abstraction of a running program and the managed data (information required to describe and manage a running program as shown in the below image)

- Memory context:
  - Text and Data, Stack and Heap
- Hardware context:
  - General purpose registers, Program Counter, Stack pointer, Stack frame pointer, …
- **OS context:**
  - **Process ID, Process State, …**

2.3 Memeory hierachy



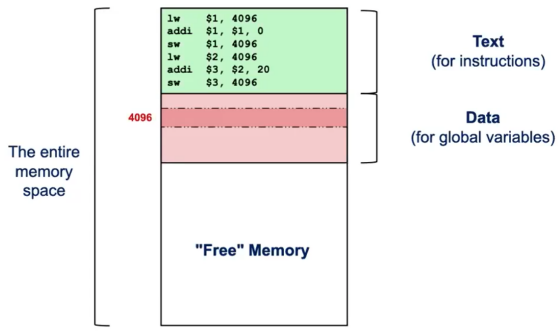
1. CPU register can be accessed within 1ns, but only has around 100 bytes(very fast but vary expensive)(directly accessed by the compiler)
2. L1,L2,L3 cache (managed by hardware)
3. DRAM(still expensive but way cheaper than that above)(managed by OS but it's accessed by load resources instruction)
4. As the memory goes done, it's slower but cheaper(time and physics) to store unit information
IMPORTANT: The register is under the control of compiler while the cache(software has nothing to do with it) is under the control of hardware; main memory is managed by OS

```
int i = 0;
  ...
i = i + 20;
```
C Program

```
lw   $1, 4096
addi $1, $0, 0
sw   $1, 4096

lw   $2, 4096
addi $3, $2, 20
sw   $3, 4096
```
Corresponding MIPS-like Assembly Code

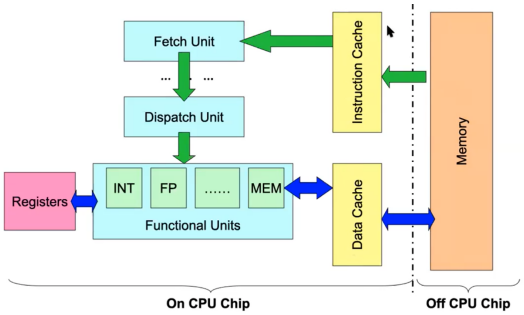
Who decide 4095 represents i: compiler can help



Operating system store text and data into the same memory

Here we assume the info is stored in the way shown in the image above
The compiler can see the size of the text
the data part can still be divided into two part(initialized data and uninitialized data)

3 computer organization review



3.1 memory

storeage for instruction and data
managed by the OS(a key resource), accesses through load/store

3.2 cache

Fast and invisible to software
Duplicate part of the memory for fast access
Usually split into instruction cache and data cache

3.3 Fetch unit

loads instruction from memory
Location indicated by special register called program counter(PC)

3.4 Functional units

carry out the instruction execution
dedicated to different instruction type

3.5 Register

Internal storeage for the fastest access speed

3.5.1 General purpose registers(GPR)

accessible by user program(i.e.visible to compiler)

3.5.2 special registers

1. program counter(PC)
2. stack pointer (SP)
3. frame pointer(FP)
4. program status word(PSW)

3.6 The step flow of a basic instruction execution

1. instruction X is fetched
memory location indicated by program counter
2. Instruction X dispatched to the corresponding functional unit
read operands usually from memory or GPR
result computed
write value if applicable(usually to memory or GPR)
3. Instruction X is completed(PC updated for the next instruction)

4 The composition of process

1. The executable
2. instruction
3. data
4. information under execution
5. Memory context(text and data; Function call; Dynamically allocated memory)
6. Hardware context(general purpose registers, program counter)
7. OS context (process state)

5 Issues regarding function call

5.1 The scope of variables

5.2 control flow

1. if function f(int a, int b) call g(int i,int j) when the g returns, how to manage the local variable and the process flow
2. Need to jump to the function body
3. need to resume when the called function return
4. Minimally, need to save the PC of the caller and restore it later

5.3 Data storage Issue

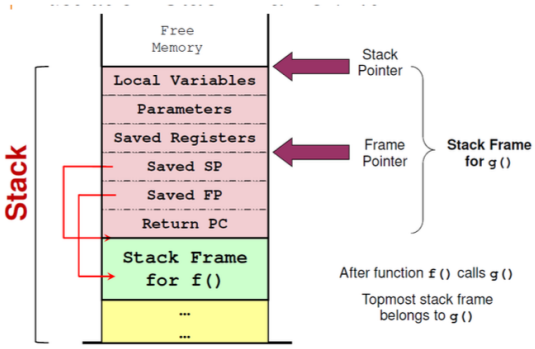
1. need to pass parameters to the function
2. need to capture the return result
3. may have local variable declaration Therefore, need a new and private region of memory that is dynamically used by the function

6 stack

6.1 definition

stack is used to store information about function invocation. The invocation is described by a stack frame.
Since stack can be of variable size, the **top** of the stack region is logically indicated by **stack pointer**. Stack pointer is usually stored in a specialised register in the CPU(the stack pointer usually store the address where the value of the stack is stored). The stack frame usually stores the following items:

1. return PC so that the caller function can be resumed after termination of callee
   PC is a register in a computer processor that contains the address (location) of the instruction being executed at the current time
2. saved FP of caller
3. saved SP of caller
4. saved registers
5. parameters
6. Local variables



6.2 Stack memory region

The new memory region to store information function invocation

6.3 stack frame v1.0

Information of a function invocation is described by a stack frame

6.3.1 Stack frame

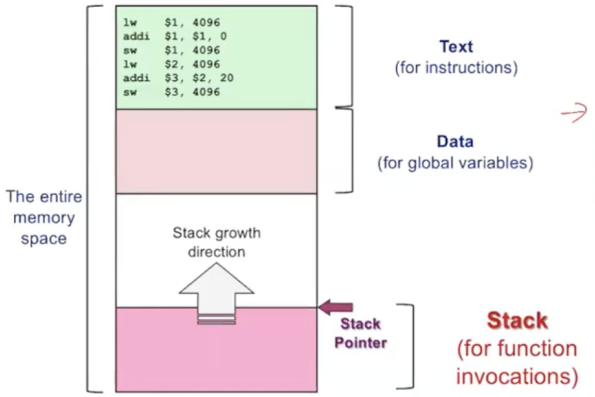
1. return address of the caller
2. arguments for the function
3. storage for local variables
4. other information

6.3.2 Stack frame is added(pushed)on top when a function is invoked(stack grows)

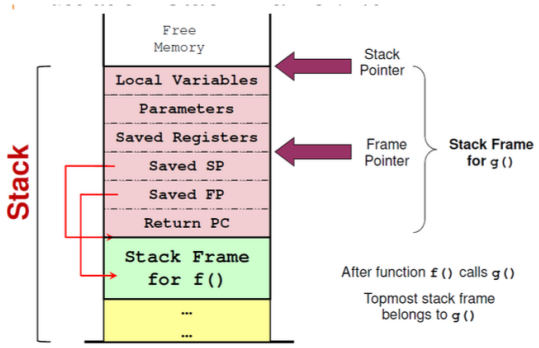
6.3.3 Stack fram is removed(poped)from top when a function call ends(stack "shrinks")

6.3.4 Stack pointer

The top of the stack region is logically indicated by a stack pointer
Most CPU has a specialized register for this purpose
Stack can grow towards higher or lower addresses(depend on platform)



6.3.5 e.g hwo the stack pointer and stack frame help with function call

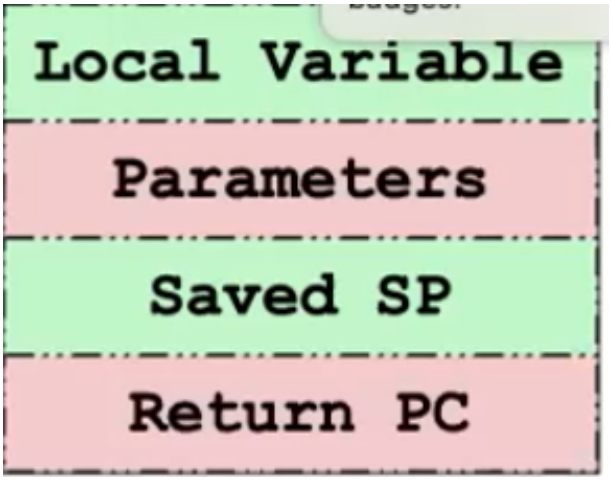


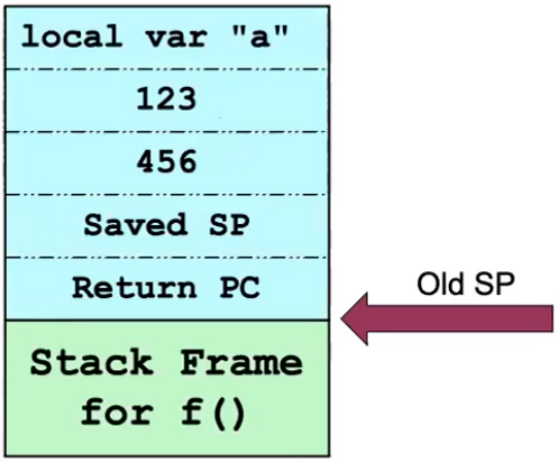
Here when the function f calls the function g, we have to create a new stack frame for g(for the local viriables and parameters etc.) hence the stack pointer is moved up to point above the new stack frame
There are different ways to setup stack frame

6.3.6 one example of call/setup

1. caller: pass parameters with registers and/or stack
2. caller: save return PC on stack

3. transfer the control from caller to callee
4. callee: save the old stack pointer(SP)
5. callee: allocate space for local variabes of callee on stack
6. callee: adjust SP to point to new stack top





6.3.7 one example of return/teardown

1. callee: place return result on stack(if applicable)
2. callee: restore saved stack pointer
3. transfer control back to caller using saved PC
4. utilize return result (if applicable)
5. continue execution in caller

6.3.8 Additional information

1. frame pointer
2. saved register

6.3.9 frame pointer

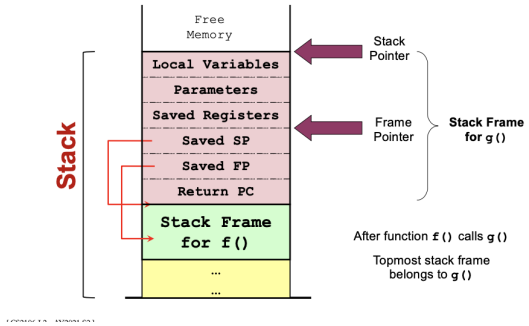
1. the frame pointer always points to a fix location in a stack frame
2. It's an optional pointer, only exists in some processors(base pointer, BP, on x86)
3. it makes other items be accessed as a displacement from the frame pointer(through load/store instructions, not push/pop)

6.3.10 Saved registers

general purpose registers(GPR)
1. GPRs on most processors are very limited(e.g MIPS has 32 GPRs, x86 has 16 GPRs)
2. When GPRs are exhausted, we will use memory to temporary hold the GPR value(1. the GPR can be reused for other purpose; the GPR value can be restored afterwards)

6.4 Stack frame v2.0

Illustration: **Stack Frame v2.0**



6.4.1 step flow of call/setup

1. caller: pass parameters with registers and/or stack
2. caller: save return PC on stack
3. transfer the control from caller to callee
4. callee: save **registers** used by callee. save old FP,SP
5. callee: allocate space for local variabes of callee on stack
6. callee: adjust SP to point to new stack top

6.4.2 step flow of return/teardown

1. callee: restore saved registers, FP
2. transfer control back to caller using saved PC
3. caller: continues execution in calller

6.5 A good question for understanding stack

What is the output of the following code fragment? You can assume that memory locations contain random values at the beginning. State your assumptions (if any).

c) 3 marks

```
void f( )
{
    int i;
    printf( "%d\n", i);
    i=1234;
}

void g( int para )
{
     printf("%d\n", para)
}
```

```
void main()
{
     f();
     f();
     g( 5432 );
     f();
}
```

Stack

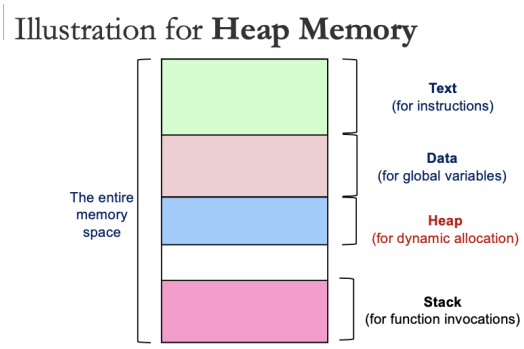
a2.png

7 Dynamically allocated memory

7.1 Many programming languages allow dynamically allocated memory

e.g.
In C: the malloc() function call
In C++: the **new** keyword
In java: the **new** keyword

7.2 The heap memory



The heap is responsible for the dynamic memory
The heap grows in the inverse direction comparing with stack

7.2.1 Not compact

A very special character about the heap memory is that: it should not be necessarily compact general(example if it initially stores A-B-C-D-E; if B is deleted, it becomes A- -C-D-E, the space occupied by B should not be filled in short time, and can be used to store data(less or equal to B) later (this also makes the memory out of order in Heap)

7.2.2 tricky heap management

Heap memory is a lot trickier to manage due to its nature:
Variable size
Variable allocation / deallocation timing
can easily construct a scenario where heap memory are allocated /deallocated in such a way to create "holes" in the memory
Free memory block squeezed in between of occupied memory block
will learn more in the memory management (much) later in the course

8 Process ID

To distinguish processes from each other

8.1 PID

process ID (PID), a number
Unique among processes

8.1.1 OS dependent issues regarding PID

1. Are PIDs reused?(linux: yes but there is a big range more likely run out of resources);
2.Does it limit the maximum no. of processes?

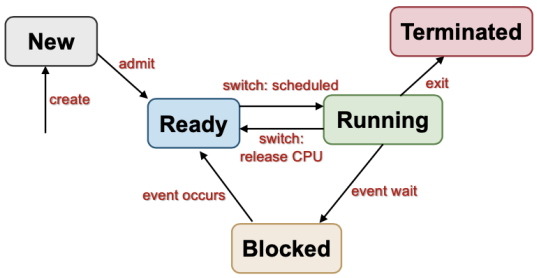
3.Are there reserved PIDs? (1 is usally the kernel)

9 Process state

The sets of process state is called process model

9.1 5-state process model

The 5-state process model is a generic and conceptual process model(not exist exactly the way in reality)



9.1.1 New

New process created
May still be under initialization, not yet ready

9.1.2 Ready

process is waiting to run

9.1.3 Running

Process being executed on CPU
transition from running to ready can be voluntary or not voluntary

9.1.4 Blocked

Process waiting (sleeping) for event
Cannot execute until event is available
From blocked state cannot go into the running state directly, have to go into the running state firstly because there is somebody running there, have to wait for the approve of OS

9.1.5 Terminated

Process has finished execution, may require OS cleanup
After that there is no proof that the process ever existing unless a process log is made

9.2 Process State Transitions in 5-Stage Mode

9.2.1 Create(nil → New)

New process is created, it's not ready for execution until every thing needed is initialized

9.2.2 Admit(New → Ready)

Process ready to be scheduled for running

9.2.3 Switch(Ready → Running)

Process selected to run

9.2.4 Switch(Running → Ready)

Process gives up CPU voluntarily or **preempted** by scheduler
Preemptive system: process can takes CPU at anytime
Non-preemptive system: the system cannot take the CPU from process unless the process approve or give up the CPU(related to scheduling)

9.2.5 Even wait(Running → Blocked)

Process requests event/resource/service which is not available/in process(System call, waiting for I/O, (more later))

9.2.6 Even wait(Blocked → Ready)

Event occurs → process can continue

9.3 Global View of Process States

Given n processes:

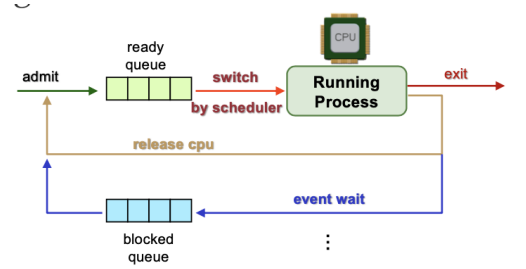
9.3.1 With 1 CPU:

$\leq$1 process in running state
In this module, more focus on i core conceptually 1 transition at a time

9.3.2 With m CPU:

$\leq$m processes in running state
possibly parallel transitions

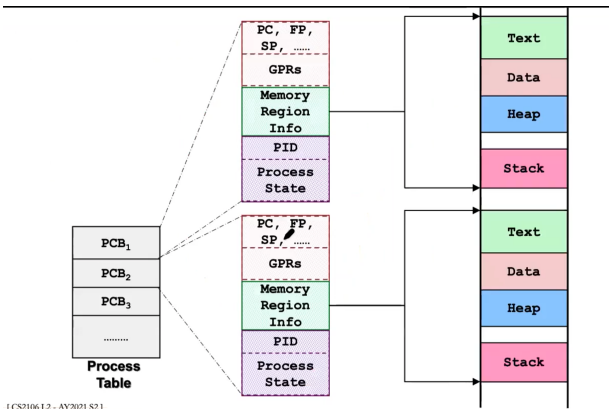
9.4 Queuing Model of 5 state transition



Notes:
•More than 1 process can be in ready + blocked queues
•May have separate event queues
•Queuing model gives global view of the processes, i.e. how the OS views them

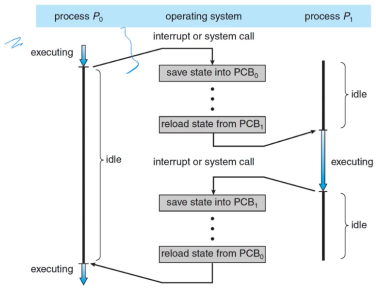
10 Process Table

The entire execution context for a process(Traditionally called Process Control Block (PCB) or Process Table Entry)
Kernel maintains PCB for all processes(Conceptually stored as one table representing all processes)



Each PCB will have a context of process
each context will heave memory regions and registers
If the process have a lot of memory, it will uses some pointer to a larger memory(not all memory are in PCB)
Different processes have different PCB(ID,value, registers and memory are different)

Context Switch



Context switch: The OS take the control of the execution of PCBs and save state of PCB(later will introduce this in details)

11 Exercise

11.1 What resources will be used during the execution of the following program

```c
#include<stdio.h>
int main(){
    printf("helloworld\n");
    return 0;
}
```

1. CPU: yes
2. IO device: yes because there is printf
3. memory: yes the program is stored in the memory
4.compiler: no, compiler is not resource; the program has to be compiled first before execution

11.2 Which system component is responsible for stack frame setup and teardown

1. programmer: yes, instruction are made by programmer and compiler
2. compiler: yes,instruction are made by programmer and compiler
3. kernel part of the OS: no, nothing to do with the OS because it's instruction that deal the pointer
4. Non-kernel part of the OS:no, nothing to do with the OS because it's instruction that deal the pointer
5. Hardware: no, the stack pointer is more higher than hardware

11.3 Who decides which variable are stored in register and which are stored in memory

1. programmer: yes, it's the instruction dealing with the register/memory
2. Compiler: yes, it's the instruction dealing with the register/memory
3. Kernel: nothing to do with OS
4. Non-kernel part of the OS: nothing to do with OS
5 hardware: higher archi than hardware

11.4 Which of the following are part of the context of a process

1. General purpose registers: yes, part of the hardware context of the process
2. Stack pointers: yes
3. Stack content: yes, part of the memory context of the process
4. Program counter: yes, save the state of the PC of the process
5. L1 instruction cache content: no, cache is not software structure; cache is hardware optimization
6. L1 data cache content: no, cache is not software structure; cache is hardware optimization
7. L2 cache content: no, cache is not software structure; cache is hardware optimization
8. Entire DRAM content: no, hardware optimization; it includes a lot of thing
9. The state of the branch predictor: no, also hardware optimization IMPORTANT: The register is under the control of compiler while the cache(software has nothing to do with it) is under the control of hardware; main memory is managed by OS